Super Pense-bête VIP : Intelligence artificielle

Afshine Amidi et Shervine Amidi

8 septembre 2019

Table des matières

1 Modèles basés sur le réflex

1.1 Prédicteurs linéaires

Dans cette section, nous allons explorer les modèles basés sur le réflex qui peuvent s'améliorer avec l'expérience s'appuyant sur des données ayant une correspondance entrée-sortie.

❒ **Vecteur caractéristique** – Le vecteur caractéristique (en anglais *feature vector*) d'une entrée x est noté $\phi(x)$ et se décompose en :

$$\boxed{\phi(x) = \begin{bmatrix} \phi_1(x) \\ \vdots \\ \phi_d(x) \end{bmatrix} \in \mathbb{R}^d}$$

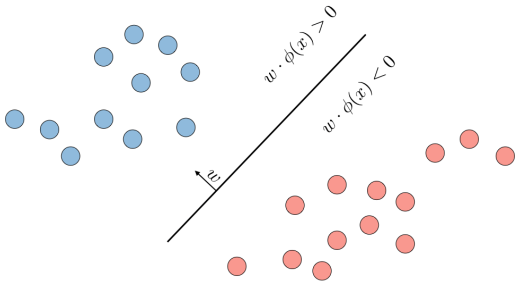
❒ **Score** – Le score $s(x,w)$ d'un exemple $(\phi(x),y) \in \mathbb{R}^d \times \mathbb{R}$ associé à un modèle linéaire de paramètres $w \in \mathbb{R}^d$ est donné par le produit scalaire :

$$\boxed{s(x,w) = w \cdot \phi(x)}$$

1.1.1 Classification

❒ **Classifieur linéaire** – Étant donnés un vecteur de paramètres $w \in \mathbb{R}^d$ et un vecteur caractéristique $\phi(x) \in \mathbb{R}^d$, le classifieur linéaire binaire f_w est donné par :

$$\boxed{f_w(x) = \text{sign}(s(x,w)) = \begin{cases} +1 & \text{if } w \cdot \phi(x) > 0 \\ -1 & \text{if } w \cdot \phi(x) < 0 \\ ? & \text{if } w \cdot \phi(x) = 0 \end{cases}}$$



❒ **Marge** – La marge (en anglais *margin*) $m(x,y,w) \in \mathbb{R}$ d'un exemple $(\phi(x),y) \in \mathbb{R}^d \times \{-1,+1\}$ associée à un modèle linéaire de paramètre $w \in \mathbb{R}^d$ quantifie la confiance associée à une prédiction : plus cette valeur est grande, mieux c'est. Cette quantité est donnée par :

$$\boxed{m(x,y,w) = s(x,w) \times y}$$

1.1.2 Régression

❒ **Régression linéaire** – Étant donnés un vecteur de paramètres $w \in \mathbb{R}^d$ et un vecteur caractéristique $\phi(x) \in \mathbb{R}^d$, le résultat d'une régression linéaire de paramètre w, notée f_w, est donné par :

$$\boxed{f_w(x) = s(x,w)}$$

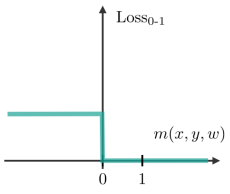
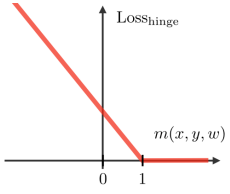
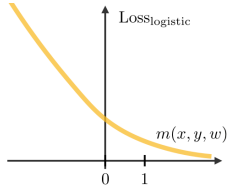
❒ **Résidu** – Le résidu $\text{res}(x,y,w) \in \mathbb{R}$ est défini comme étant la différence entre la prédiction $f_w(x)$ et la vraie valeur y :

$$\boxed{\text{res}(x,y,w) = f_w(x) - y}$$

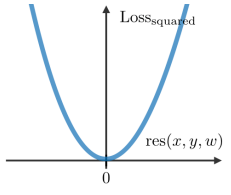
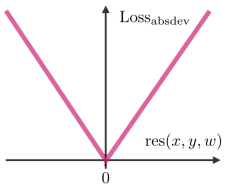
1.2 Minimisation de la fonction objectif

❒ **Fonction objectif** – Une fonction objectif (en anglais *loss function*) $\text{Loss}(x,y,w)$ traduit notre niveau d'insatisfaction avec les paramètres w du modèle dans la tâche de prédiction de la sortie y à partir de l'entrée x. C'est une quantité que l'on souhaite minimiser pendant la phase d'entraînement.

❒ **Cas de la classification** – Trouver la classe d'un exemple x appartenant à $y \in \{-1,+1\}$ peut être faite par le biais d'un modèle linéaire de paramètre w à l'aide du prédicteur $f_w(x) \triangleq \text{sign}(s(x,w))$. La qualité de cette prédiction peut alors être évaluée au travers de la marge $m(x,y,w)$ intervenant dans les fonctions objectif suivantes :

Fonction objectif	Zéro-un	Hinge	Logistique
$\text{Loss}(x,y,w)$	$1_{\{m(x,y,w) \leqslant 0\}}$	$\max(1 - m(x,y,w), 0)$	$\log(1 + e^{-m(x,y,w)})$
Illustration			

❒ **Cas de la régression** – Prédire la valeur $y \in \mathbb{R}$ associée à l'exemple x peut être faite par le biais d'un modèle linéaire de paramètre w à l'aide du prédicteur $f_w(x) \triangleq s(x,w)$. La qualité de cette prédiction peut alors être évaluée au travers du résidu $\text{res}(x,y,w)$ intervenant dans les fonctions objectif suivantes :

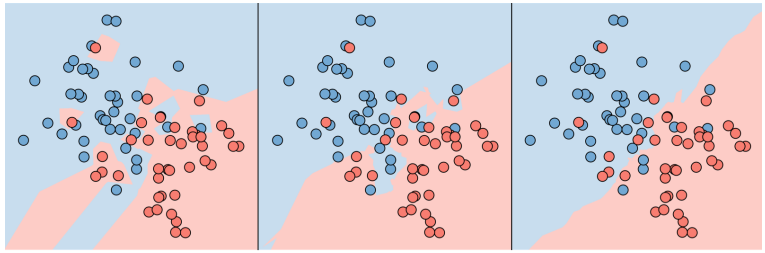
Nom	Erreur quadratique	Erreur absolue
$\text{Loss}(x,y,w)$	$(\text{res}(x,y,w))^2$	$\lvert\text{res}(x,y,w)\rvert$
Illustration		

❒ **Processus de minimisation de la fonction objectif** – Lors de l'entraînement d'un modèle, on souhaite minimiser la valeur de la fonction objectif évaluée sur l'ensemble d'entraînement :

$$\boxed{\text{TrainLoss}(w) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \text{Loss}(x,y,w)}$$

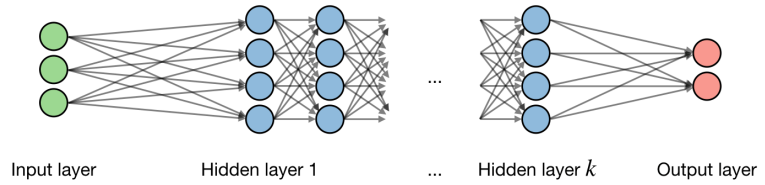
1.3 Prédicteurs non linéaires

❒ **k plus proches voisins** – L'algorithme des k plus proches voisins (en anglais *k-nearest neighbors* ou *k-NN*) est une approche non paramétrique où la réponse associée à un exemple est déterminée par la nature de ses k plus proches voisins de l'ensemble d'entraînement. Cette démarche peut être utilisée pour la classification et la régression.



Remarque : plus le paramètre k est grand, plus le biais est élevé. À l'inverse, la variance devient plus élevée lorsque l'on réduit la valeur k.

❒ **Réseaux de neurones** – Les réseaux de neurones (en anglais *neural networks*) constituent un type de modèle basés sur des couches (en anglais *layers*). Parmi les types de réseaux populaires, on peut compter les réseaux de neurones convolutionnels et récurrents (abbréviés respectivement en CNN et RNN *en anglais*). Une partie du vocabulaire associé aux réseaux de neurones est détaillée dans la figure ci-dessous :



En notant i la i-ème couche du réseau et j son j-ième neurone, on a :

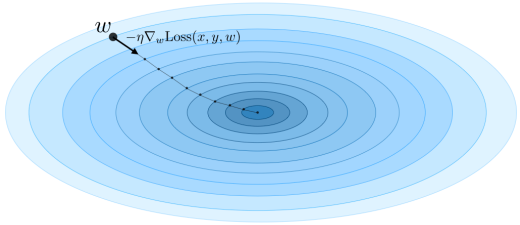
$$\boxed{z_j^{[i]} = {w_j^{[i]}}^T x + b_j^{[i]}}$$

où l'on note w, b, x, z le coefficient, le biais ainsi que la variable de sortie respectivement.

1.4 Algorithme du gradient stochastique

❒ **Descente de gradient** – En notant $\eta \in \mathbb{R}$ le taux d'apprentissage (en anglais learning rate ou step size), la règle de mise à jour des coefficients pour cet algorithme utilise la fonction objectif Loss(x,y,w) de la manière suivante :

$$\boxed{w \longleftarrow w - \eta\nabla_w \text{Loss}(x,y,w)}$$



❒ **Mises à jour stochastiques** – L'algorithme du gradient stochastique (en anglais *stochastic gradient descent* ou *SGD*) met à jour les paramètres du modèle en parcourant les exemples $(\phi(x),y) \in \mathcal{D}_{\text{train}}$ de l'ensemble d'entraînement un à un. Cette méthode engendre des mises à jour rapides à calculer mais qui manquent parfois de robustesse.

❒ **Mises à jour par lot** – L'algorithme du gradient par lot (en anglais *batch gradient descent* ou *BGD*) met à jour les paramètre du modèle en utilisant des lots entiers d'exemples (e.g. la totalité de l'ensemble d'entraînement) à la fois. Cette méthode calcule des directions de mise à jour des coefficients plus stable au prix d'un plus grand nombre de calculs.

1.5 Peaufinage de modèle

❒ **Classe d'hypothèses** – Une classe d'hypothèses $\mathcal{F}$ est l'ensemble des prédicteurs candidats ayant un $\phi(x)$ fixé et dont le paramètre w peut varier :

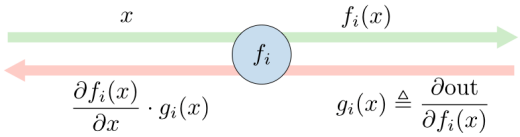
$$\boxed{\mathcal{F} = \left\{f_w : w \in \mathbb{R}^d\right\}}$$

❒ **Fonction logistique** – La fonction logistique σ, aussi appelée en anglais *sigmoid function*, est définie par :

$$\boxed{\forall z \in]-\infty,+\infty[, \quad \sigma(z) = \frac{1}{1+e^{-z}}}$$

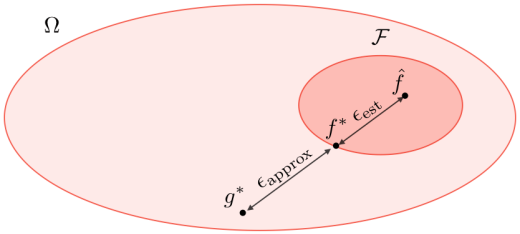
Remarque : la dérivée de cette fonction s'écrit $\sigma'(z) = \sigma(z)(1-\sigma(z))$.

❒ **Rétropropagation du gradient (en anglais *backpropagation*)** – La propagation avant (en anglais *forward pass*) est effectuée via f_i, valeur correspondant à l'expression appliquée à l'étape i. La propagation de l'erreur vers l'arrière (en anglais *backward pass*) se fait via $g_i = \frac{\partial \text{out}}{\partial f_i}$ et décrit la manière dont f_i agit sur la sortie du réseau.

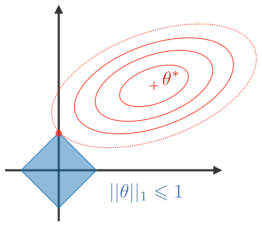
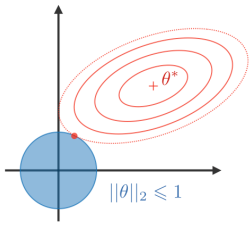
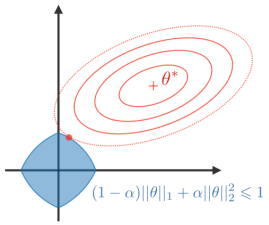


❒ **Erreur d'approximation et d'estimation** – L'erreur d'approximation ϵ_{approx} représente la distance entre la classe d'hypothèses $\mathcal{F}$ et le prédicteur optimal g^*. De son côté, l'erreur

d'estimation ϵ_{est} quantifie la qualité du prédicteur $\hat{f}$ par rapport au meilleur prédicteur f^* de la classe d'hypothèses $\mathcal{F}$.



❒ **Régularisation** – Le but de la régularisation est d'empêcher le modèle de surapprendre (en anglais *overfit*) les données en s'occupant ainsi des problèmes de variance élevée. La table suivante résume les différents types de régularisation couramment utilisés :

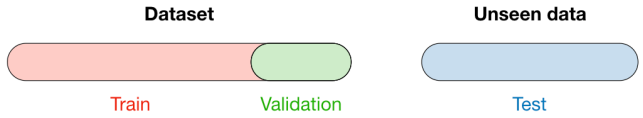
LASSO	Ridge	Elastic Net
- Réduit les coefficients à 0 - Bénéfique pour la sélection de variables	Rapetissit les coefficients	Compromis entre sélection de variables et coefficients de faible magnitude
		
$\ldots + \lambda\|\|\theta\|\|_1$ $\lambda \in \mathbb{R}$	$\ldots + \lambda\|\|\theta\|\|_2^2$ $\lambda \in \mathbb{R}$	$\ldots + \lambda\left[(1-\alpha)\|\|\theta\|\|_1 + \alpha\|\|\theta\|\|_2^2\right]$ $\lambda \in \mathbb{R}$, $\alpha \in [0,1]$

❒ **Hyperparamètres** – Les hyperparamètres sont les paramètres de l'algorithme d'apprentissage et incluent parmi d'autres le type de caractéristiques utilisé ainsi que le paramètre de régularisation λ, le nombre d'itérations T le taux d'apprentissage η.

❒ **Vocabulaire** – Lors de la sélection d'un modèle, on divise les données en 3 différentes parties :

Données d'entraînement	Données de validation	Données de test
- Le modèle y est entrainé - Constitue normalement 80 du jeu de données	- Le modèle y est évalué - Constitue normalement 20 du jeu de données - Aussi appelé données de développement (en anglais *hold-out* ou *development set*)	- Le modèle y donne ses prédictions - Données jamais observées

Une fois que le modèle a été choisi, il est entrainé sur le jeu de données entier et testé sur l'ensemble de test (qui n'a jamais été vu). Ces derniers sont représentés dans la figure ci-dessous :



1.6 Apprentissage non supervisé

Les méthodes d'apprentissage non supervisé visent à découvrir la structure (parfois riche) des données.

1.6.1 k-moyennes (en anglais k-*means*)

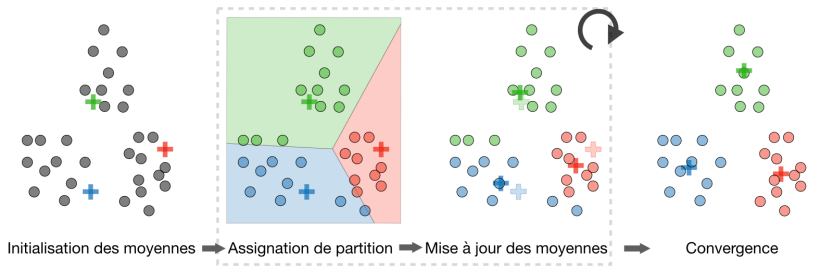
❒ **Partitionnement** – Étant donné un ensemble d'entraînement $\mathcal{D}_{\text{train}}$, le but d'un algorithme de partitionnement (en anglais *clustering*) est d'assigner chaque point $\phi(x_i)$ à une partition $z_i \in \{1,\ldots,k\}$.

❒ **Fonction objectif** – La fonction objectif d'un des principaux algorithmes de partitionnement, k-moyennes, est donné par :

$$\boxed{\text{Loss}_{\text{k-means}}(x,\mu) = \sum_{i=1}^{n} ||\phi(x_i) - \mu_{z_i}||^2}$$

❒ **Algorithme** – Après avoir aléatoirement initialisé les centroïdes de partitions $\mu_1,\mu_2,\ldots,\mu_k \in \mathbb{R}^n$, l'algorithme k-moyennes répète l'étape suivante jusqu'à convergence :

$$\boxed{z_i = \arg\min_j ||\phi(x_i) - \mu_j||^2} \quad \text{and} \quad \boxed{\mu_j = \frac{\sum_{i=1}^{m} 1_{\{z_i=j\}}\phi(x_i)}{\sum_{i=1}^{m} 1_{\{z_i=j\}}}}$$



1.6.2 Analyse des composantes principales

❒ **Valeur propre, vecteur propre** – Étant donnée une matrice $A \in \mathbb{R}^{n\times n}$, λ est dite être une valeur propre de A s'il existe un vecteur $z \in \mathbb{R}^n \backslash \{0\}$, appelé vecteur propre, tel que :

$$\boxed{Az = \lambda z}$$

❒ **Théorème spectral** – Soit $A \in \mathbb{R}^{n\times n}$. Si A est symétrique, alors A est diagonalisable par une matrice réelle orthogonale $U \in \mathbb{R}^{n\times n}$. En notant $\Lambda = \text{diag}(\lambda_1,...,\lambda_n)$, on a :

$$\boxed{\exists \Lambda \text{ diagonal}, \quad A = U\Lambda U^T}$$

Remarque : le vecteur propre associé à la plus grande valeur propre est appelé le vecteur propre principal de la matrice A.

❒ **Algorithme** – La procédure d'analyse des composantes principales (en anglais *Principal Component Analysis* ou *PCA*) est une technique de réduction de dimension qui projette les données sur k dimensions en maximisant la variance des données de la manière suivante :

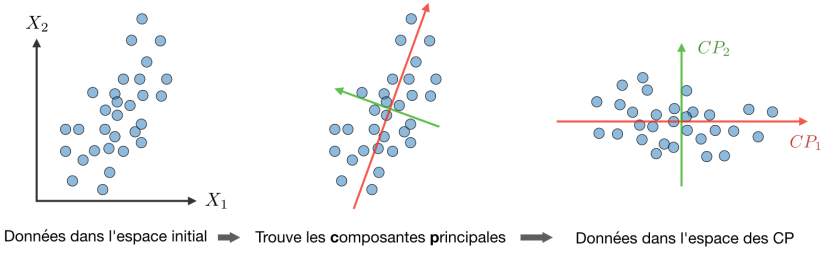
— Étape 1 : Normaliser les données pour avoir une moyenne de 0 et un écart-type de 1.

$$\boxed{x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{\sigma_j}} \quad \text{where} \quad \boxed{\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}} \quad \text{and} \quad \boxed{\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2}$$

— Étape 2 : Calculer $\Sigma = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}x^{(i)^T} \in \mathbb{R}^{n\times n}$, qui est symétrique avec des valeurs propres réelles.

— Étape 3 : Calculer $u_1, ..., u_k \in \mathbb{R}^n$ les k valeurs propres principales orthogonales de Σ, i.e. les vecteurs propres orthogonaux des k valeurs propres les plus grandes.

— Étape 4 : Projeter les données sur $\text{span}_{\mathbb{R}}(u_1,...,u_k)$. Cette procédure maximise la variance sur tous les espaces à k dimensions.



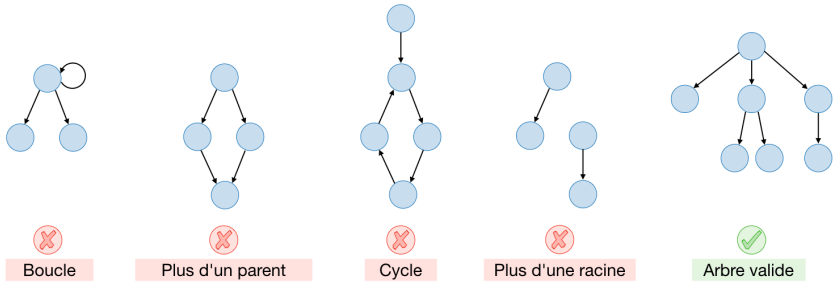
2 Modèles basés sur les états

2.1 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).

2.1.1 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 :

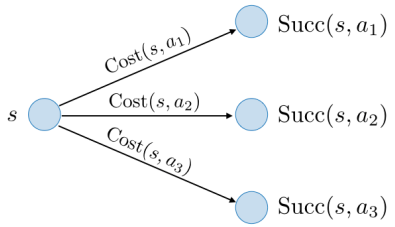
— un état de départ s_{start}

— 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)

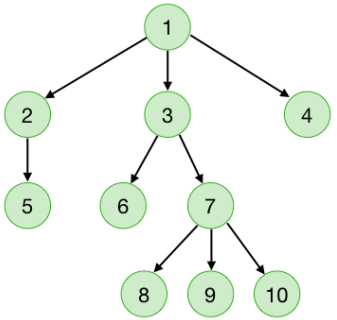


L'objectif est de trouver un chemin minimisant le coût total des actions utilisées.

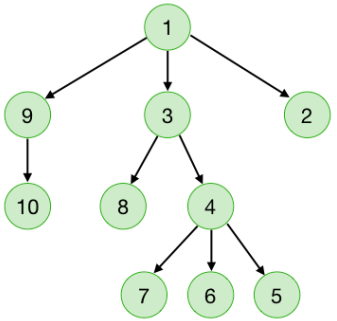
❒ **Retour sur trace** – L'algorithme de retour sur trace (en anglais *backtracking search*) est un algorithme récursif explorant naïvement toutes les possibilités jusqu'à trouver le chemin de coût minimal. Ici, le coût des actions peut aussi bien être positif que négatif.

❒ **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 $c \geqslant 0$.

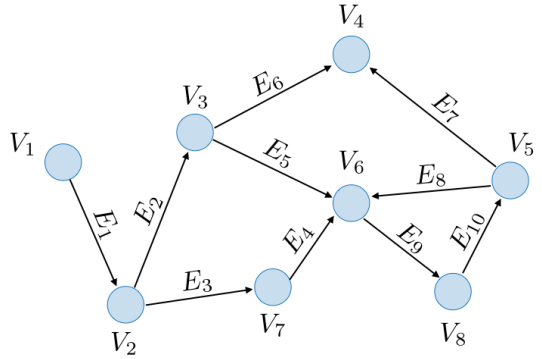
❒ **Récapitulatif des algorithmes de parcours d'arbre** – En notant b le nombre d'actions par état, d la profondeur de la solution et D la profondeur maximale, on a :

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	$c \geqslant 0$	$\mathcal{O}(d)$	$\mathcal{O}(b^d)$

2.1.2 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).

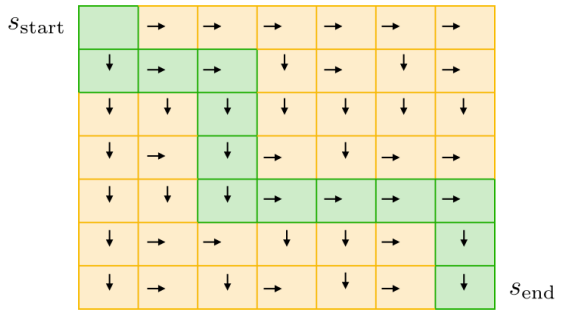


Remarque : un graphe est dit être acyclique lorsqu'il ne contient pas de cycle.

❒ **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 :

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

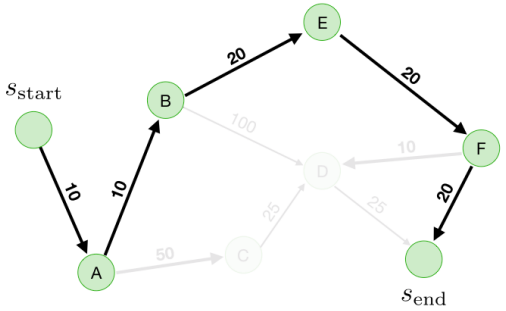


Remarque : la figure ci-dessus illustre une approche ascendante alors que la formule nous donne l'intuition d'une résolution avec une approche descendante.

❒ **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.

2.1.3 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 :

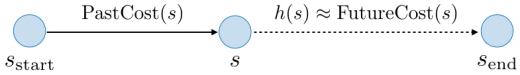
— fait décroître le coût estimé de chaque état-action du vrai chemin minimisant y donné par la base d'apprentissage,

— 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.

2.1.4 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 :

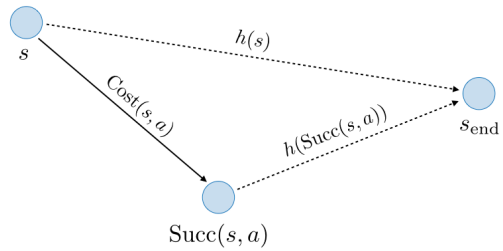
$$\boxed{\text{Cost}'(s,a) = \text{Cost}(s,a) + h(\text{Succ}(s,a)) - h(s)}$$

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 :

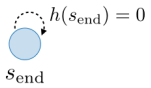
— Pour tous états s et actions a,

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



— 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 :

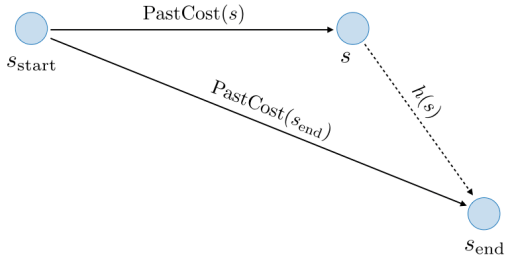
$$\boxed{h(s) \leqslant \text{FutureCost}(s)}$$

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

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

❒ **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.

2.1.5 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 :

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

❒ **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 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 :

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

2.2 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.

2.2.1 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}

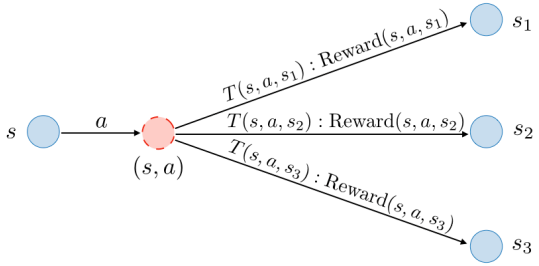
— l'ensemble des actions Actions(s) pouvant être effectuées à partir de l'état s

— la probabilité de transition $T(s,a,s')$ de l'état s vers l'état s' après avoir pris l'action a

— la récompense 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 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 :

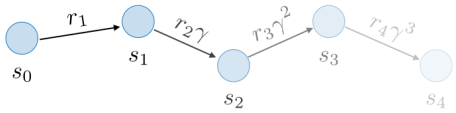
$$\forall s,a, \quad \boxed{\sum_{s' \in \text{ States}} T(s,a,s') = 1}$$

❒ **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.

2.2.2 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

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

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 :

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

❒ **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 :

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

❒ **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

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

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.

2.2.3 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 :

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

❒ **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 :

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

2.3 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.

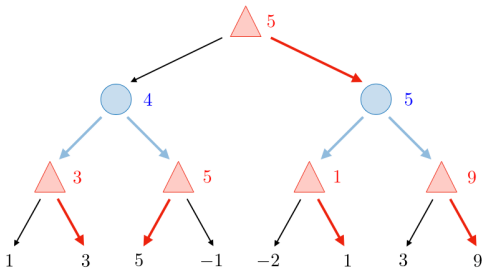
❒ **Types de politiques** – Il y a deux types de politiques :

- 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 :

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

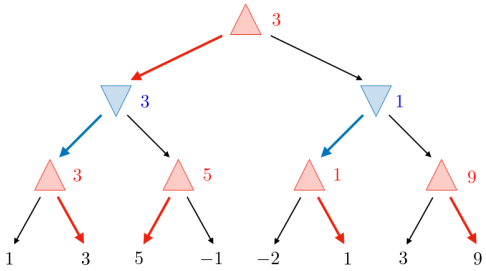
Remarque : expectimax est l'analogue de l'algorithme d'itération sur la valeur pour les MDPs.



❒ **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 :

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

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.

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

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

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

— *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.

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

À la fin, on a la relation suivante :

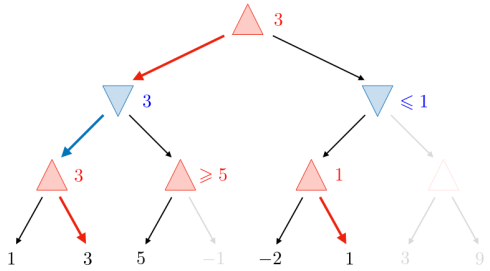
$$\boxed{V(\pi_{\text{exptmax}}, \pi_{\min}) \leqslant V(\pi_{\max}, \pi_{\min}) \leqslant V(\pi_{\max}, \pi) \leqslant V(\pi_{\text{exptmax}}, \pi)}$$

2.3.1 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 $\text{FutureCost}(s)$.

❒ **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 :

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

2.3.2 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 :

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

2.3.3 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.

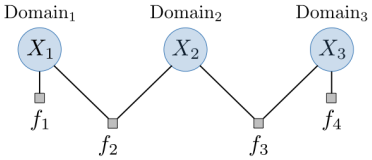
3 Modèles basés sur les variables

3.1 Problèmes de satisfaction de contraintes

Dans cette section, notre but est de trouver des affectations de poids maximisants dans des problèmes impliquant des modèles basés sur les variables. Un avantage comparé aux modèles basés sur les états est que ces algorithmes sont plus commodes lorsqu'il s'agit de transcrire des contraintes spécifiques à certains problèmes.

3.1.1 Graphes de facteurs

❒ **Définition** – Un graphe de facteurs, aussi appelé champ aléatoire de Markov, est un ensemble de variables $X = (X_1,...,X_n)$ où $X_i \in \text{Domain}_i$ muni de m facteurs $f_1,...,f_m$ où chaque $f_j(X) \geqslant 0$.



❒ **Arité** – Le nombre de variables dépendant d'un facteur f_j est appelé son arité.

Remarque : les facteurs d'arité 1 et 2 sont respectivement appelés unaire et binaire.

❒ **Affectation de poids** – Chaque affectation $x = (x_1,...,x_n)$ donne un poids $\text{Weight}(x)$ défini comme étant le produit de tous les facteurs f_j appliqués à cette affectation. Son expression est donnée par :

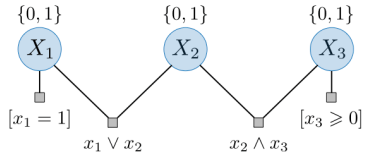
$$\boxed{\text{Weight}(x) = \prod_{j=1}^{m} f_j(x)}$$

❒ **Problème de satisfaction de contraintes** – Un problème de satisfaction de contraintes (en anglais *constraint satisfaction problem* ou *CSP*) est un graphe de facteurs où tous les facteurs sont binaires ; on les appelle "contraintes".

$$\boxed{\forall j \in [\![1,m]\!], \quad f_j(x) \in \{0,1\}}$$

Ici, on dit que l'affectation x satisfait la contrainte j si et seulement si $f_j(x) = 1$.

❒ **Affectation consistante** – Une affectation x d'un CSP est dite consistante si et seulement si $\text{Weight}(x) = 1$, i.e. toutes les contraintes sont satisfaites.



3.1.2 Mise en ordre dynamique

❒ **Facteurs dépendants** – L'ensemble des facteurs dépendants de la variable X_i dont l'affectation partielle est x est appelé $D(x,X_i)$ et désigne l'ensemble des facteurs liant X_i à des variables déjà affectées.

❒ **Recherche avec retour sur trace** – L'algorithme de recherche avec retour sur trace (en anglais *backtracking search*) est utilisé pour trouver l'affectation de poids maximum d'un graphe de facteurs. À chaque étape, une variable non assignée est choisie et ses valeurs sont explorées par récursivité. On peut utiliser un processus de mise en ordre dynamique sur le choix des variables et valeurs et/ou d'anticipation (i.e. élimination précoce d'options non consistantes) pour explorer le graphe de manière plus efficace. La complexité temporelle dans tous les cas reste néanmoins exponentielle : $O(|\text{Domain}|^n)$.

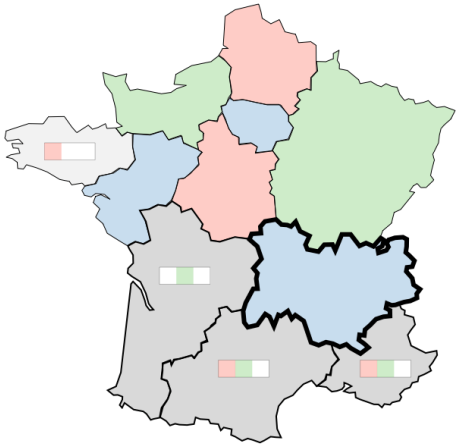
❒ **Vérification en avant** – La vérification en avant (*forward checking* en anglais) est une heuristique d'anticipation à une étape qui enlève des variables voisines les valeurs impossibles de manière préemptive. Cette méthode a les caractéristiques suivantes :

- Après l'affectation d'une variable X_i, les valeurs non consistantes sont éliminées du domaine de tous ses voisins.
- Si l'un de ces domaines devient vide, la recherche locale s'arrête.
- Si l'on enlève l'affectation d'une valeur X_i, on doit restaurer le domaine de ses voisins.

❒ **Variable la plus contrainte** – L'heuristique de la variable la plus contrainte (en anglais *most constrained variable* ou *MCV*) sélectionne la prochaine variable sans affectation ayant le moins de valeurs consistantes. Cette procédure a pour effet de faire échouer les affectations impossibles plus tôt dans la recherche, permettant un élagage plus efficace.

❒ **Valeur la moins contraignante** – L'heuristique de la valeur la moins contraignante (en anglais *least constrained value* ou *LCV*) sélectionne pour une variable donnée la prochaine valeur maximisant le nombre de valeurs consistantes chez les variables voisines. De manière intuitive, on peut dire que cette procédure choisit en premier les valeurs qui sont le plus susceptible de marcher.

Remarque : en pratique, cette heuristique est utile quand tous les facteurs sont des contraintes.



L'exemple ci-dessus est une illustration du problème de coloration de graphe à 3 couleurs en utilisant l'algorithme de recherche avec retour sur trace couplé avec les heuristiques de MCV, de LCV ainsi que de vérification en avant à chaque étape.

❒ **Arc-consistance** – On dit que l'arc-consistance de la variable X_l par rapport à X_k est vérifiée lorsque pour tout $x_l \in \text{Domain}_l$:

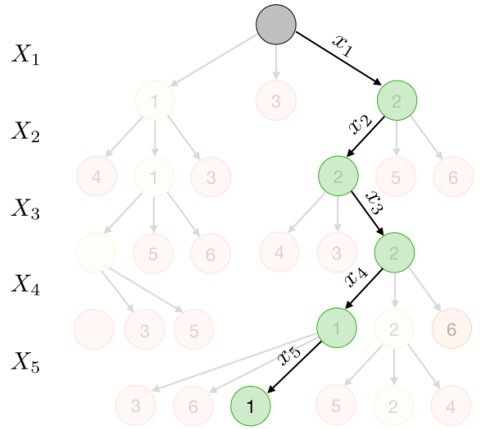
- les facteurs unaires de X_l sont non-nuls,
- il existe au moins un $x_k \in \text{Domain}_k$ tel que n'importe quel facteur entre X_l et X_k est non nul.

❒ **AC-3** – L'algorithme d'AC-3 est une heuristique qui applique le principe de vérification en avant à toutes les variables susceptibles d'être concernées. Après l'affectation d'une variable, cet algorithme effectue une vérification en avant et applique successivement l'arc-consistance avec tous les voisins de variables pour lesquels le domaine change.

Remarque : AC-3 peut être codé de manière itérative ou récursive.

3.1.3 Méthodes approximatives

❒ **Recherche en faisceau** – L'algorithme de recherche en faisceau (en anglais *beam search*) est une technique approximative qui étend les affectations partielles de n variables de facteur de branchement $b = |\text{Domain}|$ en explorant les K meilleurs chemins qui s'offrent à chaque étape. La largeur du faisceau $K \in \{1,...,b^n\}$ détermine la balance entre efficacité et précision de l'algorithme. Sa complexité en temps est de $O(n \cdot Kb\log(Kb))$.
L'exemple ci-dessous illustre une recherche en faisceau de paramètres $K = 2$, $b = 3$ et $n = 5$.



Remarque : $K = 1$ correspond à la recherche gloutonne alors que $K \to +\infty$ est équivalent à effectuer un parcours en largeur.

❒ **Modes conditionnels itérés** – L'algorithme des modes conditionnels itérés (en anglais *iterated conditional modes* ou *ICM*) est une technique itérative et approximative qui modifie l'affectation d'un graphe de facteurs une variable à la fois jusqu'à convergence. À l'étape i, X_i prend la valeur v qui maximise le produit de tous les facteurs connectés à cette variable.

Remarque : il est possible qu'ICM reste bloqué dans un minimum local.

❒ **Échantillonnage de Gibbs** – La méthode d'échantillonnage de Gibbs (en anglais *Gibbs sampling*) est une technique itérative et approximative qui modifie les affectations d'un graphe de facteurs une variable à la fois jusqu'à convergence. À l'étape i :

- on assigne à chaque élément $u \in \text{Domain}_i$ un poids $w(u)$ qui est le produit de tous les facteurs connectés à cette variable,
- on échantillonne v de la loi de probabilité engendrée par w et on l'associe à X_i.

Remarque : la méthode d'échantillonnage de Gibbs peut être vue comme étant la version probabiliste de ICM. Cette méthode a l'avantage de pouvoir échapper aux potentiels minimum locaux dans la plupart des situations.

3.1.4 Transformations sur les graphes de facteurs

❒ **Indépendance** – Soit A,B une partition des variables X. On dit que A et B sont indépendants s'il n'y a pas d'arête connectant A et B et on écrit :

$$\boxed{A \text{ et } B \text{ indépendants } \iff A \perp\!\!\!\perp B}$$

Remarque : l'indépendance est une propriété importante car elle nous permet de décomposer la situation en sous-problèmes que l'on peut résoudre en parallèle.

❒ **Indépendance conditionnelle** – On dit que A et B ont conditionnellement indépendants par rapport à C si le fait de conditionner sur C produit un graphe dans lequel A et B sont indépendants. Dans ce cas, on écrit :

$$\boxed{A \text{ et } B \text{ cond. indép. par rapport à } C \iff A \perp\!\!\!\perp B|C}$$

❒ **Conditionnement** – Le conditionnement est une transformation visant à rendre des variables indépendantes et ainsi diviser un graphe de facteurs en pièces plus petites qui peuvent être traitées en parallèle et utiliser le retour sur trace. Pour conditionner par rapport à une variable $X_i = v$, on :

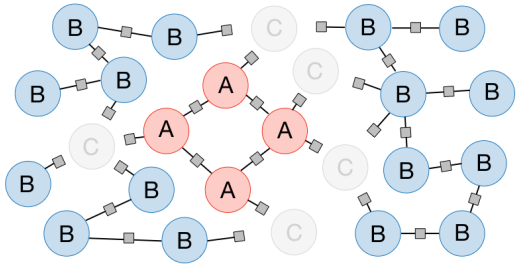
— considère toues les facteurs f_1,...,f_k qui dépendent de X_i
— enlève X_i et f_1,...,f_k
— ajoute $g_j(x)$ pour $j \in \{1,...,k\}$ défini par :

$$\boxed{g_j(x) = f_j(x \cup \{X_i : v\})}$$

❒ **Couverture de Markov** – Soit $A \subseteq X$ une partie des variables. On définit MarkovBlanket(A) comme étant les voisins de A qui ne sont pas dans A.

❒ **Proposition** – Soit C = MarkovBlanket(A) et $B = X \backslash (A \cup C)$. On a alors :

$$\boxed{A \perp\!\!\!\perp B|C}$$



❒ **Élimination** – L'élimination est une transformation consistant à enlever X_i d'un graphe de facteurs pour ensuite résoudre un sous-problème conditionné sur sa couverture de Markov où l'on :

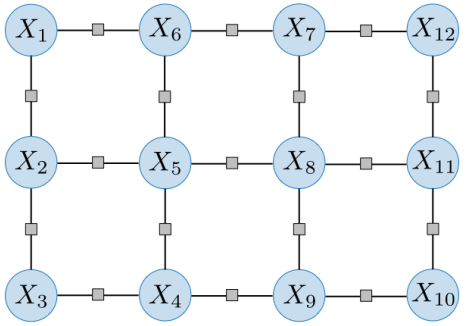
— considère tous les facteurs $f_{i,1}$,...,$f_{i,k}$ qui dépendent de X_i
— enlève X_i et $f_{i,1}$,...,$f_{i,k}$
— ajoute $f_{\text{new},i}(x)$ défini par :

$$\boxed{f_{\text{new},i}(x) = \max_{x_i} \prod_{l=1}^{k} f_{i,l}(x)}$$

❒ **Largeur arborescente** – La largeur arborescente (en anglais *treewidth*) d'un graphe de facteurs est l'arité maximum de n'importe quel facteur créé par élimination avec le meilleur ordre de variable. En d'autres termes,

$$\boxed{\text{Treewidth} = \min_{\text{orderings}} \max_{i \in \{1,...,n\}} \text{arity}(f_{\text{new},i})}$$

L'exemple ci-dessous illustre le cas d'un graphe de facteurs ayant une largeur arborescente égale à 3.



Remarque : trouver le meilleur ordre de variable est un problème NP-difficile.

3.2 Réseaux bayésiens

Dans cette section, notre but est de calculer des probabilités conditionnelles. Quelle est la probabilité d'un événement étant donné des observations ?

3.2.1 Introduction

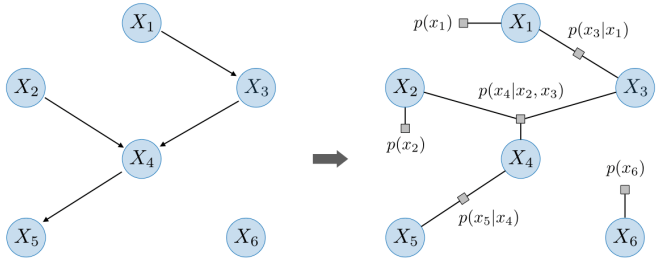
❒ **Explication** – Supposons que les causes C_1 et C_2 influencent un effet E. Le conditionnement sur l'effet E et une des causes (disons C_1) change la probabilité de l'autre cause (disons C_2). Dans ce cas, on dit que C_1 a expliqué C_2.

❒ **Graphe orienté acyclique** – Un graphe orienté acyclique (en anglais *directed acyclic graph* ou *DAG*) est un graphe orienté fini sans cycle orienté.

❒ **Réseau bayésien** – Un réseau bayésien (en anglais *Bayesian network*) est un DAG qui définit une loi de probabilité jointe sur les variables aléatoires $X = (X_1,...,X_n)$ comme étant le produit des lois de probabilités conditionnelles locales (une pour chaque nœud) :

$$\boxed{P(X_1 = x_1,...,X_n = x_n) \triangleq \prod_{i=1}^{n} p(x_i|x_{\text{Parents}(i)})}$$

Remarque : les réseaux bayésiens sont des graphes de facteurs imprégnés de concepts de probabilité.



❒ **Normalisation locale** – Pour chaque $x_{\text{Parents}(i)}$, tous les facteurs sont localement des lois de probabilité conditionnelles. Elles doivent donc vérifier :

$$\boxed{\sum_{x_i} p(x_i|x_{\text{Parents}(i)}) = 1}$$

De ce fait, les sous-réseaux bayésiens et les distributions conditionnelles sont consistants.

Remarque : les lois locales de probabilité conditionnelles sont de vraies lois de probabilité conditionnelles.

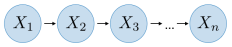
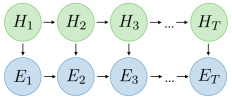
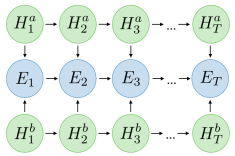
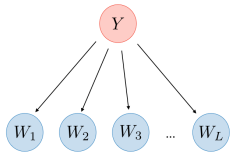
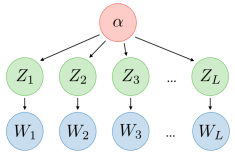
❒ **Marginalisation** – La marginalisation d'un nœud sans enfant entraine un réseau bayésian sans ce nœud.

3.2.2 Programmes probabilistes

❒ **Concept** – Un programme probabiliste rend aléatoire l'affectation de variables. De ce fait, on peut imaginer des réseaux bayésiens compliqués pour la génération d'affectations sans avoir à écrire de manière explicite les probabilités associées.

Remarque : quelques exemples de programmes probabilistes incluent parmi d'autres le modèle de Markov caché (en anglais hidden Markov model ou HMM), HMM factoriel, le modèle bayésien naïf (en anglais naive Bayes), l'allocation de Dirichlet latente (en anglais latent Dirichlet allocation ou LDA), le modèle à blocs stochastiques (en anglais stochastic block model).

❒ **Récapitulatif** – La table ci-dessous résume les programmes probabilistes les plus fréquents ainsi que leur champ d'application associé :

Programme	Algorithme	Illustration	Exemple
Modèle de Markov	$X_i \sim p(X_i\|X_{i-1})$		Modélisation du langage
Modèle de Markov caché (HMM)	$H_t \sim p(H_t\|H_{t-1})$ $E_t \sim p(E_t\|H_t)$		Suivi d'objet
HMM factoriel	$H_t^o \underset{o\in\{a,b\}}{\sim} p(H_t^o\|H_{t-1}^o)$ $E_t \sim p(E_t\|H_t^a, H_t^b)$		Suivi de plusieurs objets
Bayésien naïf	$Y \sim p(Y)$ $W_i \sim p(W_i\|Y)$		Classification de document
Allocation de Dirichlet latente (LDA)	$\alpha \in \mathbb{R}^K$ distribution $Z_i \sim p(Z_i\|\alpha)$ $W_i \sim p(W_i\|Z_i)$		Modélisation de sujet

3.2.3 Inférence

❒ **Stratégie générale pour l'inférence probabiliste** – La stratégie que l'on utilise pour calculer la probabilité $P(Q|E=e)$ d'une requête Q étant donnée l'observation $E=e$ est la suivante :

- Étape 1 : on enlève les variables qui ne sont pas les ancêtres de la requête Q ou de l'observation E par marginalisation
- Étape 2 : on convertit le réseau bayésien en un graphe de facteurs
- Étape 3 : on conditionne sur l'observation $E=e$
- Étape 4 : on enlève les nœuds déconnectés de la requête Q par marginalisation
- Étape 5 : on lance un algorithme d'inférence probabiliste (manuel, élimination de variables, échantillonnage de Gibbs, filtrage particulaire)

❒ **Algorithme progressif-rétrogressif** – L'algorithme progressif-rétrogressif (en anglais *forward-backward*) calcule la valeur exacte de $P(H=h_k|E=e)$ pour chaque $k \in \{1,...,L\}$ dans le cas d'un HMM de taille L. Pour ce faire, on procède en 3 étapes :

- Étape 1 : pour $i \in \{1,...,L\}$, calculer $F_i(h_i) = \sum_{h_{i-1}} F_{i-1}(h_{i-1})p(h_i|h_{i-1})p(e_i|h_i)$
- Étape 2 : pour $i \in \{L,...,1\}$, calculer $B_i(h_i) = \sum_{h_{i+1}} B_{i+1}(h_{i+1})p(h_{i+1}|h_i)p(e_{i+1}|h_{i+1})$
- Étape 3 : pour $i \in \{1,...,L\}$, calculer $S_i(h_i) = \frac{F_i(h_i)B_i(h_i)}{\sum_{h_i} F_i(h_i)B_i(h_i)}$

avec la convention $F_0 = B_{L+1} = 1$. À partir de cette procédure et avec ces notations, on obtient

$$\boxed{P(H=h_k|E=e) = S_k(h_k)}$$

Remarque : cet algorithme interprète une affectation comme étant un chemin où chaque arête $h_{i-1} \to h_i$ a un poids $p(h_i|h_{i-1})p(e_i|h_i)$.

❒ **Échantillonnage de Gibbs** – L'algorithme d'échantillonnage de Gibbs (en anglais *Gibbs sampling*) est une méthode itérative et approximative qui utilise un petit ensemble d'affectations (particules) pour représenter une loi de probabilité. Pour une affectation aléatoire x, l'échantillonnage de Gibbs effectue les étapes suivantes pour $i \in \{1,...,n\}$ jusqu'à convergence :

— Pour tout $u \in \text{Domain}_i$, on calcule le poids $w(u)$ de l'affectation x où $X_i = u$

— On échantillonne v de la loi de probabilité engendrée par w : $v \sim P(X_i = v | X_{-i} = x_{-i})$

— On pose $X_i = v$

Remarque : X_{-i} veut dire $X \backslash \{X_i\}$ et x_{-i} représente l'affectation correspondante.

❒ **Filtrage particulaire** – L'algorithme de filtrage particulaire (en anglais *particle filtering*) approxime la densité postérieure de variables d'états à partir des variables observées en suivant K particules à la fois. En commençant avec un ensemble de particules C de taille K, on répète les 3 étapes suivantes :

— Étape 1 : proposition - Pour chaque particule $x_{t-1} \in C$, on échantillonne x avec loi de probabilité $p(x|x_{t-1})$ et on ajoute x à un ensemble C'.

— Étape 2 : pondération - On associe chaque x de l'ensemble C' au poids $w(x) = p(e_t|x)$, où e_t est l'observation vue à l'instant t.

— Étape 3 : é-échantillonnage - On échantillonne K éléments de l'ensemble C' en utilisant la loi de probabilité engendrée par w et on les met dans C : ce sont les particules courantes x_t.

Remarque : une version plus coûteuse de cet algorithme tient aussi compte des particules passée à l'étape de proposition.

❒ **Maximum de vraisemblance** – Si l'on ne connaît pas les lois de probabilité locales, on peut les trouver en utilisant le maximum de vraisemblance.

$$\boxed{\max_{\theta} \prod_{x \in \mathcal{D}_{\text{train}}} p(X = x; \theta)}$$

❒ **Lissage de Laplace** – Pour chaque loi de probabilité d et affectation partielle $(x_{\text{Parents}(i)}, x_i)$, on ajoute λ à $\text{count}_d(x_{\text{Parents}(i)}, x_i)$ et on normalise ensuite pour obtenir des probabilités.

❒ **Espérance-maximisation** – L'algorithme d'espérance-maximisation (en anglais *expectation-maximization* ou *EM*) est une méthode efficace utilisée pour estimer le paramètre θ via l'estimation du maximum de vraisemblance en construisant de manière répétée une borne inférieure de la vraisemblance (étape E) et en optimisant cette borne inférieure (étape M) :

— Étape E : on évalue la probabilité postérieure $q(h)$ que chaque point e vienne d'une partition particulière h avec :

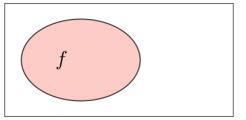
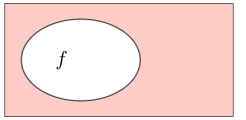
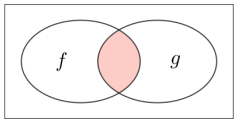
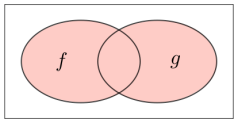
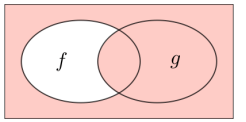
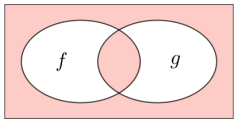
$$\boxed{q(h) = P(H = h | E = e; \theta)}$$

— Étape M : on utilise la probabilité postérieure $q(h)$ en tant que poids de la partition h sur les points e pour déterminer θ through via le maximum de vraisemblance.

4 Modèles basés sur la logique

4.1 Bases

❒ **Syntaxe de la logique propositionnelle** – En notant f et g formules et $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ opérateurs, on peut écrire les expressions logiques suivantes :

Nom	Symbole	Signification	Illustration
Affirmation	f	f	
Négation	$\neg f$	non f	
Conjonction	$f \wedge g$	f et g	
Disjonction	$f \vee g$	f ou g	
Implication	$f \rightarrow g$	si f alors g	
Biconditionnel	$f \leftrightarrow g$	f, c'est à dire g	

Remarque : n'importe quelle formule peut être construite de manière récursive à partir de ces opérateurs.

❒ **Modèle** – Un modèle w dénote une combinaison de valeurs binaires liées à des symboles propositionnels.

Exemple : l'ensemble de valeurs de vérité $w = \{A : 0, B : 1, C : 0\}$ est un modèle possible pour les symboles propositionnels A, B and C.

❒ **Interprétation** – L'interprétation $\mathcal{I}(f,w)$ outputs whether model w satisfies formula f :

$$\boxed{\mathcal{I}(f,w) \in \{0,1\}}$$

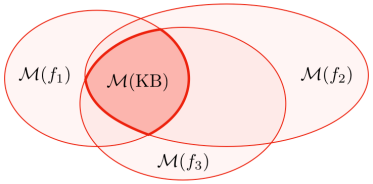
❒ **Ensemble de modèles** – $\mathcal{M}(f)$ dénote l'ensemble des modèles w qui satisfont la formule f. Sa définition mathématique est donnée par :

$$\boxed{\forall w \in \mathcal{M}(f), \quad \mathcal{I}(f,w) = 1}$$

4.2 Base de connaissance

❒ **Définition** – La base de connaissance KB est la conjonction de toutes les formules considérées jusqu'à présent. L'ensemble des modèles de la base de connaissance est l'intersection de l'ensemble des modèles satisfaisant chaque formule. En d'autres termes :

$$\boxed{\mathcal{M}(\text{KB}) = \bigcap_{f \in \text{KB}} \mathcal{M}(f)}$$



❒ **Interprétation en termes de probabilités** – La probabilité que la requête f soit évaluée à 1 peut être vue comme la proportion des modèles w de la base de connaissance KB qui satisfait f, i.e. :

$$\boxed{P(f|\text{KB}) = \frac{\displaystyle\sum_{w \in \mathcal{M}(\text{KB}) \cap \mathcal{M}(f)} P(W = w)}{\displaystyle\sum_{w \in \mathcal{M}(\text{KB})} P(W = w)}}$$

❒ **Satisfaisabilité** – La base de connaissance KB est dite satisfaisable si au moins un modèle w satisfait toutes ses contraintes. En d'autres termes :

$$\boxed{\text{KB satisfaisable} \iff \mathcal{M}(\text{KB}) \neq \varnothing}$$

Remarque : $\mathcal{M}(KB)$ dénote l'ensemble des modèles compatibles avec toutes les contraintes de la base de connaissance.

❒ **Relation entre formules et base de connaissance** – On définit les propriétés suivantes entre la base de connaissance KB et une nouvelle formule f :

Nom	Formulation mathématique	Illustration	Notes
KB déduit f	$\mathcal{M}(\text{KB}) \cap \mathcal{M}(f) = \mathcal{M}(\text{KB})$		- f n'apporte aucune nouvelle information - Aussi écrit KB $\models f$
KB contredit f	$\mathcal{M}(\text{KB}) \cap \mathcal{M}(f) = \varnothing$		- Aucun modèle ne satisfait les contraintes après l'ajout de f - Équivalent à KB $\models \neg f$
f est contingent à KB	$\mathcal{M}(\text{KB}) \cap \mathcal{M}(f) \neq \varnothing$ et $\mathcal{M}(\text{KB}) \cap \mathcal{M}(f) \neq \mathcal{M}(\text{KB})$		- f ne contredit pas KB - f ajoute une quantité d'information non triviale à KB

❒ **Vérification de modèles** – Un algorithme de vérification de modèles (*model checking* en anglais) prend comme argument une base de connaissance KB et nous renseigne si celle-ci est satisfaisable ou pas.

Remarque : DPLL et WalkSat sont des exemples populaires d'algorithmes de vérification de modèles.

❒ **Règle d'inférence** – Une règle d'inférence de prémisses $f_1,...,f_k$ et de conclusion g s'écrit :

$$\boxed{\frac{f_1,...,f_k}{g}}$$

❒ **Algorithme de chaînage avant** – Partant d'un ensemble de règles d'inférence Rules, l'algorithme de chaînage avant (en anglais *forward inference algorithm*) parcourt tous les $f_1,...,f_k$ et ajoute g à la base de connaissance KB si une règle parvient à une telle conclusion. Cette démarche est répétée jusqu'à ce qu'aucun autre ajout ne puisse être fait à KB.

❒ **Dérivation** – On dit que KB dérive f (noté KB $\vdash f$) par le biais des règles Rules soit si f est déjà dans KB ou si elle se fait ajouter pendant l'application du chaînage avant utilisant les règles Rules.

❒ **Propriétés des règles d'inférence** – Un ensemble de règles d'inférence Rules peut avoir les propriétés suivantes :

Name	Formulation mathématique	Notes
Validité	$\{f : \text{KB} \vdash f\} \subseteq \{f : \text{KB} \models f\}$	- Les formules inférées sont déduites par KB - Peut être vérifiée une règle à la fois - "*Rien que la vérité*"
Complétude	$\{f : \text{KB} \vdash f\} \supseteq \{f : \text{KB} \models f\}$	- Les formules déduites par KB sont soit déjà dans la base de connaissance, soit inférées de celle-ci - "*La vérité dans sa totalité*"

4.3 Logique propositionnelle

Dans cette section, nous allons parcourir les modèles logiques utilisant des formules logiques et des règles d'inférence. L'idée est de trouver le juste milieu entre expressivité et efficacité.

❒ **Clause de Horn** – En notant p_1,...,p_k et q des symboles propositionnels, une clause de Horn s'écrit :

$$\boxed{(p_1 \wedge ... \wedge p_k) \longrightarrow q}$$

Remarque : quand q = false, cette clause de Horn est "négative", autrement elle est appelée "stricte".

❒ **Modus ponens** – Sur les symboles propositionnels f_1,...,f_k et p, la règle de modus ponens est écrite :

$$\boxed{\frac{f_1,...,f_k, \quad (f_1 \wedge ... \wedge f_k) \longrightarrow p}{p}}$$

Remarque : l'application de cette règle se fait en temps linéaire, puisque chaque exécution génère une clause contenant un symbole propositionnel.

❒ **Complétude** – Modus ponens est complet lorsqu'on le munit des clauses de Horn si l'on suppose que KB contient uniquement des clauses de Horn et que p est un symbole propositionnel qui est déduit. L'application de modus ponens dérivera alors p.

❒ **Forme normale conjonctive** – La forme normale conjonctive (en anglais *conjunctive normal form* ou *CNF*) d'une formule est une conjonction de clauses, chacune d'entre elles étant une disjonction de formules atomiques.

Remarque : en d'autres termes, les CNFs sont des $\wedge$ de $\vee$.

❒ **Représentation équivalente** – Chaque formule en logique propositionnelle peut être écrite de manière équivalente sous la forme d'une formule CNF. Le tableau ci-dessous présente les propriétés principales permettant une telle conversion :

Nom de la règle		Début	Résultat
Élimine	$\leftrightarrow$	$f \leftrightarrow g$	$(f \rightarrow g) \wedge (g \rightarrow f)$
	$\rightarrow$	$f \rightarrow g$	$\neg f \vee g$
	$\neg\neg$	$\neg\neg f$	f
Distribue	$\neg$ sur $\wedge$	$\neg(f \wedge g)$	$\neg f \vee \neg g$
	$\neg$ sur $\vee$	$\neg(f \vee g)$	$\neg f \wedge \neg g$
	$\vee$ sur $\wedge$	$f \vee (g \wedge h)$	$(f \vee g) \wedge (f \vee h)$

❒ **Règle de résolution** – Pour des symboles propositionnels f_1,...,f_n, et g_1,...,g_m ainsi que p, la règle de résolution s'écrit :

$$\boxed{\frac{f_1 \vee ... \vee f_n \vee p, \quad \neg p \vee g_1 \vee ... \vee g_m}{f_1 \vee ... \vee f_n \vee g_1 \vee ... \vee g_m}}$$

Remarque : l'application de cette règle peut prendre un temps exponentiel, vu que chaque itération génère une clause constituée d'une partie des symboles propositionnels.

❒ **Inférence basée sur la règle de résolution** – L'algorithme d'inférence basée sur la règle de résolution se déroule en plusieurs étapes :

- Étape 1 : Conversion de toutes les formules vers leur forme CNF
- Étape 2 : Application répétée de la règle de résolution
- Étape 3 : Renvoyer "non satisfaisable" si et seulement si False est dérivé

4.4 Calcul des prédicats du premier ordre

L'idée ici est d'utiliser des variables et ainsi permettre une représentation des connaissances plus compacte.

❒ **Modèle** –Un modèle w en calcul des prédicats du premier ordre lie :

- des symboles constants à des objets
- des prédicats à n-uplets d'objets

❒ **Clause de Horn** – En notant x_1,...,x_n variables et a_1,...,a_k,b formules atomiques, une clause de Horn pour le calcul des prédicats du premier ordre a la forme :

$$\boxed{\forall x_1,...,\forall x_n, \quad (a_1 \wedge ... \wedge a_k) \rightarrow b}$$

❒ **Substitution** – Une substitution θ lie les variables aux termes et Subst(θ,f) désigne le résultat de la substitution θ sur f.

❒ **Unification** – Une unification prend deux formules f et g et renvoie la substitution θ la plus générale les rendant égales :

$$\boxed{\text{Unify}[f,g] = \theta} \quad \text{t.q.} \quad \boxed{\text{Subst}[\theta,f] = \text{Subst}[\theta,g]}$$

Note : Unify[f,g] renvoie Fail si un tel θ n'existe pas.

❒ **Modus ponens** – En notant x_1,...,x_n variables, a_1,...,a_k et a'_1,...,a'_k formules atomiques et en notant θ = Unify($a'_1 \wedge ... \wedge a'_k, a_1 \wedge ... \wedge a_k$), modus ponens pour le calcul des prédicats du premier ordre s'écrit :

$$\boxed{\frac{a'_1,...,a'_k \quad \forall x_1,...,\forall x_n (a_1 \wedge ... \wedge a_k) \rightarrow b}{\text{Subst}[\theta, b]}}$$

❒ **Complétude** – Modus ponens est complet pour le calcul des prédicats du premier ordre lorsqu'il agit uniquement sur les clauses de Horn.

❒ **Règle de résolution** – En notant $f_1, ..., f_n$, $g_1, ..., g_m$, p, q formules et en posant θ = Unify(p,q), le règle de résolution pour le calcul des prédicats du premier ordre s'écrit :

$$\boxed{\frac{f_1 \vee ... \vee f_n \vee p, \quad \neg q \vee g_1 \vee ... \vee g_m}{\text{Subst}[\theta, f_1 \vee ... \vee f_n \vee g_1 \vee ... \vee g_m]}}$$

❒ **Semi-décidabilité** – Le calcul des prédicats du premier ordre, même restreint aux clauses de Horn, n'est que semi-décidable.

- si KB $\models f$, l'algorithme de chaînage avant sur des règles d'inférence complètes prouvera f en temps fini
- si KB $\not\models f$, aucun algorithme ne peut le prouver en temps fini