



UNIVERSITÉ LIBRE DE BRUXELLES

INFO-F311
INTELLIGENCE ARTIFICIELLE

Synthèse IA

Étudiants :

Rayan CONTULIANO BRAVO

Enseignants :

T. LENAERTS

5 janvier 2024

Contents

1	Introduction	2
2	Recherche Non-Informée	6
2.1	Problème de recherches	6
2.2	Graphe d'espace d'état	7
2.3	Arbres de Recherches	8
2.3.1	Recherche dans un arbre de recherche	9
2.4	Depth-First Search (DFS)	10
2.5	Breadth-First Search (BFS)	10
2.6	Iterative deepening	10
2.7	UCS	11
3	Recherche Informée	12
3.1	Greedy Best-First Search	12
3.2	A*	13
3.3	Créer des Heuristiques admissibles	14
4	Recherche Locale	15
5	Recherche Adversarial	15
5.1	Définition d'un jeu	15
5.2	Minimax	15
5.3	Alpha-Beta pruning	18
5.4	Expectimax	19
5.5	Monte Carlo Tree Search (MCTS)	21
6	Probabilités	23
6.1	Quelques définitions	23
6.2	Rappel proba	23
6.3	Inférence	26
6.3.1	Inférence par Énumération	26
6.4	Indépendance	27
7	Réseaux Bayesiens	28
7.1	Inférence	30
7.1.1	Inférence par énumération	30
7.1.1.1	Elimination de variables	31
7.2	D-Séparation	32
7.3	Echantillonage	33
7.3.1	Prior	33
7.3.2	Rejection Sampling	34
7.3.3	Likelihood Weighting	34
7.3.4	Gibbs Sampling	35
8	Modèle de Markov	36
8.1	Le modèle qu'on ne trouve pas	36

8.2 Inférence	37
8.3 Chaines de Markov	38
9 Réseaux de décisions	38
9.1 Value of Perfect Information	39
10 Markovian Decision Processes	40
10.1 Hyper-Paramètres	42
10.2 Equation de Bellman	43
10.2.1 Quantité optimale	43
10.2.2 Value Iteration	43
10.2.3 Policy Extraction	44
10.2.4 Policy Evaluation	44
10.2.5 Policy Iteration	44
11 Reinforcement Learning	45
11.1 Passive Reinforcement Learning	45
11.1.1 Model-based Passive RL	46
11.1.2 Model-free Passive RL	46
11.1.2.1 Direct evaluation	46
11.1.2.2 Temporal Difference Value Learning	47
11.1.3 Sample-based Policy Evaluation	48
11.2 Active Reinforcement Learning	48
11.2.1 Q-learning	49
11.3 Exploration vs Exploitation	50
11.3.1 ϵ -greedy	50
11.3.2 Fonctions d'exploration	50
11.3.3 Regret	51
11.4 Approximate Q-learning	51
12 Machine Learning	52
12.1 Classification	52
12.2 Model-based classification	52
12.2.1 Naïve Bayes Model	53
12.2.2 Inference for Naïve Bayes	54

1 Introduction

Définition 1.1. Qu'est-ce que l'ia

L'intelligence artificielle est une branche de l'informatique qui crée des systèmes qui pensent de manière **rationnelle**

Définition 1.2. Décisions rationnelles

Penser de manière rationnelle signifie qu'on va se concentrer sur le **choix de décisions** qui maximisent la probabilité d'atteindre un objectif donné. On va faire agir les systèmes de manière **optimale**

Remarques:

1. Être rationnel signifie donc **maximiser** l'utilité attendue.
2. On définit un objectif par son **utilité**.

Définition 1.3. Agent

Un agent est un système qui perçoit son environnement par des **capteurs** et agit sur celui-ci par des **effecteurs**.

Définition 1.4. Agent rationnel

Un agent rationnel est un agent qui agit de manière à maximiser son utilité attendue.

Remarque: Les **capteurs**, **effecteurs** et **l'environnement** permettent à l'agent de percevoir et d'agir sur le monde de manière **rationnelle**. L'**agent** est le système qui prend les décisions.

Définition 1.5. Fonction agent

La fonction agent est une fonction qui prend en entrée une séquence de perceptions et retourne une action. $f : \mathcal{P}^* \rightarrow \mathcal{AP}$

Exemple:

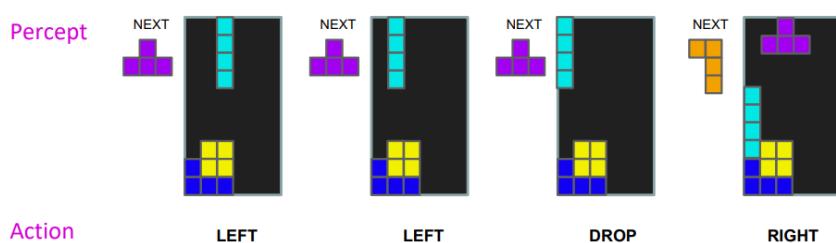


FIGURE 1.1 – Représentation d'une fonction agent dans le jeu Tetris. En fonction d'une certaine situation dans le jeu, l'agent va choisir une action.

Définition 1.6. Programme Agent

Un programme agent l est exécuté sur une machine M afin d'implémenter la fonction agent f .

Remarque: Les machines dans le monde réel sont **imparfaites** et **limitées** en vitesse d'exécution et en mémoire.

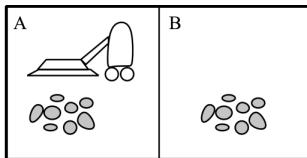


FIGURE 1.2 – Etat de l'environnement de l'aspirateur. Localisation de l'aspirateur et si un case contient de la poussière ou non

Exemple: Nous pouvons représenter un aspirateur comme un agent qui perçoit son environnement par des capteurs et agit sur celui-ci par des effecteurs.

- **Perception** : capteurs qui détectent la saleté et sa localisation dans l'espace
- **Action** : effecteurs qui déplacent l'aspirateur dans l'espace et aspire ou non

En imaginant la situation en figure 1.2, nous pouvons définir la fonction agent de l'aspirateur comme suit :

TABLE 1 – Fonction agent de l'aspirateur

Sequence de perception	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [B, Clean]	Left
[A, Clean], [B, Dirty]	Suck
etc...	etc...

Pour que notre agent soit bien rationnel, il nous faut une manière de **mesurer la performance** de celui-ci. Pour cela, nous allons définir une **fonction de performance** qui va mesurer la qualité des actions de l'agent.



Il est important de bien définir la fonction de performance car elle va déterminer le comportement de l'agent. Il se peut qu'en fonction de la manière dont vous accordé des récompenses à votre agent, celui-ci ne se comporte pas comme vous le souhaitez. Il pourrait par exemple décider de ne rien faire et de rester immobile pour maximiser son utilité attendue. Ou bien de faire tout le temps la même action car celle-ci lui rapporte le plus de points.

Exemple: On peut lui faire gagner des points ou bien lui en retirer en fonction d'une action. Comme pour éduquer un animal.

De cette manière, l'agent va savoir quelles actions lui permettent de **maximiser** son utilité attendue.

Afin de bien déterminer un environnement, les particularités de notre agents, il nous faut, **avant toute chose**, définir **PEAS**

Définition 1.7. PEAS

- **Performance** : mesure de la qualité des actions de l'agent
- **Environnement** : type d'environnement dans lequel l'agent va évoluer
- **Actuateurs** : les effecteurs de l'agent
- **Sensors** : les capteurs de l'agent



FIGURE 1.3 – Environnement Pacman

Exemple: Pour l'environnement Pacman de la figure 1.3, nous pouvons définir PEAS comme suit :

- **Performance** : -1/pas, +10/nourriture, +500/partie gagnées, -500/mort, +200/tuer un fantôme effrayé
- **Environnement** : labyrinthe **dynamique** de pacman
- **Actuateurs** : Haut, Bas, Gauche, Droite
- **Capteurs** : L'état entier visible

Définition 1.8. Types d'environnement

Il y a plusieurs types d'environnement :

- **Mono-agent** : un seul agent
- **Multi-agent** : plusieurs agents qui maximisent leur **propre** tâche (coop ou concurrentiel)
- **Déterministe** : l'état de l'environnement est déterminé **seulement** par les actions de l'agent
- **Stochastique** : l'environnement est non déterministe. L'état de l'environnement est déterminé par les actions de l'agent et par des facteurs aléatoires
- **Épisodique** : les actions de l'agent n'affectent pas les actions futures
- **Séquentiel** : les actions de l'agent affectent les actions futures
- **Dynamique** : l'environnement peut changer pendant que l'agent réfléchit
- **Statique** : l'environnement ne change pas pendant que l'agent réfléchit
- **Complètement observable** : les capteurs de l'agent perçoivent l'état complet de l'environnement
- **Partiellement observable** : les capteurs de l'agent perçoivent une partie de l'état de l'environnement
- **Discret** : un nombre fini d'états
- **Continu** : un nombre infini d'états
- **Connu** : l'agent connaît les lois de l'environnement

Il existe plusieurs types d'agents qui répondent à des environnements plus complexes :

- **Agent réflexe simple** : l'agent choisit son action en fonction de la **dernière** perception

- **Agent réflexe basé sur un modèle** : l'agent choisit son action en fonction de la dernière perception et d'un **état interne**(dépend de l'**historique** des perceptions)
- **Agent fondés sur des buts** : l'agent choisit son action en fonction de la dernière perception ainsi que des infos relatives à l'objectif
- **Agent fondés sur l'utilité** : l'agent choisit son action en fonction de sa satisfaction par rapport à l'état résultant

2 Recherche Non-Informée

Définition 2.1. Recherche non-informée

La recherche **non-informée** est une stratégie de recherche qui n'utilise **pas** d'information sur l'état de l'environnement afin de le **guider** pour trouver une solution. Elle explore simplement l'espace de recherche de manière systématique. En utilisant souvent des *algorithmes* comme **DFS**, **BFS**.

Remarque: La recherche non-informée est utilisée quand on ne connaît pas l'état de l'environnement. Lorsqu'on ne peut quantifier la qualité d'un état en utilisant des **informations heuristiques**

Définition 2.2. Agent de Plannification

Les agents de planification font des **hypothèses** sur les conséquences des actions entreprises et utilisent un **modèle** de l'environnement pour trouver un plan qui atteint son objectif.

Résolution de problèmes par la recherche :

1. **Formulation de l'objectif** : L'agent doit avoir un objectif afin de pouvoir organiser son comportement. Ca permet de limiter l'espace de recherche (*actions entreprises*)
2. **Formulation du problème** : L'agent doit avoir un moyen de représenter les actions et les états afin de pouvoir les manipuler
3. **Recherche de la solution** : Avant d'agir dans le monde réel, l'agent fait une simulation de séquences d'actions dans son modèle de l'environnement jusqu'à trouver une séquence qui mène à l'objectif. C'est la *solution*
4. **Exécution de la solution** : L'agent exécute la séquence d'actions dans le monde réel

• **Note:-**

Un plan est une séquence d'actions qui mène à l'objectif.

2.1 Problème de recherches

Définition 2.3. Problème de Recherche

Un problème de recherche est défini par :

- **Ensemble d'État** S : Une situation dans lequel l'environnement peut être agencé
- **État initial** s_0 : l'état dans lequel le problème commence
- **Actions** $A(s)$: les actions possibles dans l'état s
- **Modèle de Transition** $Result(s, a)$: la fonction qui définit les conséquences des actions. L'état résultant de l'action a dans l'état s
- **État final** : l'état que l'on veut atteindre
- **Cout de l'action** $c(s, a, s')$: le coût de l'action a dans l'état s qui mène à l'état s'
- **Solution** : Une séquence d'actions qui mène de l'état initial à l'état final. La solution peut être **optimale**, c'est à dire qu'il n'y a pas de solution dont le coût est moindre.

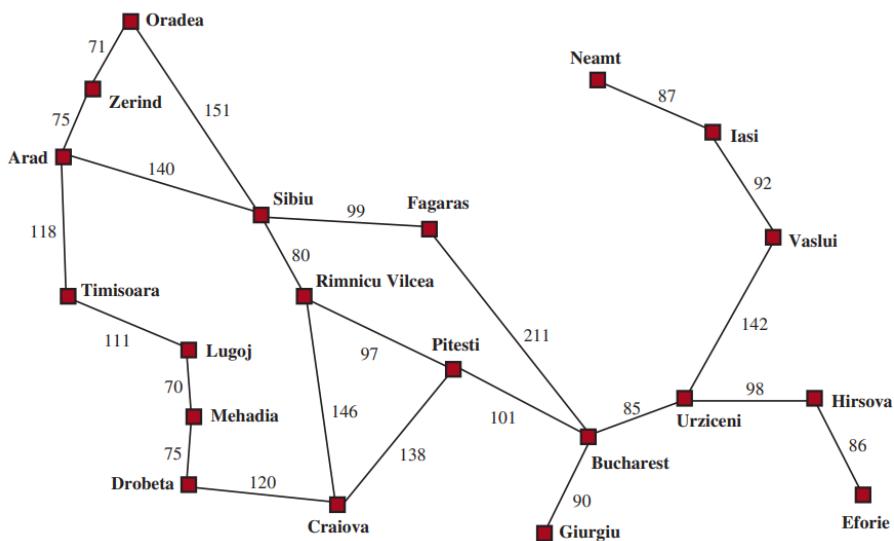


FIGURE 2.1 – Représentation simple de la Roumanie en graphe

Exemple: Nous allons dès lors modéliser le problème de trouver un chemin entre deux villes en Roumanie.

- **États** : les villes de Roumanie
- **État initial** : Arad
- **Actions** : les routes entre les villes adjacentes
- **Modèle de transition** : Atteindre une ville adjacente
- **Cout de l'action** : distance entre les villes
- **État final** : Bucharest

2.2 Graphe d'espace d'état

Définition 2.4. Graphe d'espace d'état

Un graphe d'espace d'état est un graphe qui représente les états et les actions possibles.

- **Noeuds** : les états
- **Arêtes** : les résultats d'actions, successeurs

L'état initial est le noeud racine et l'état final est un noeud (ou plusieurs ?). Les noeuds sont les représentations d'une configuration de l'environnement.



Dans ce genre de graphe, chaque état n'est représenté qu'**une seule fois**.

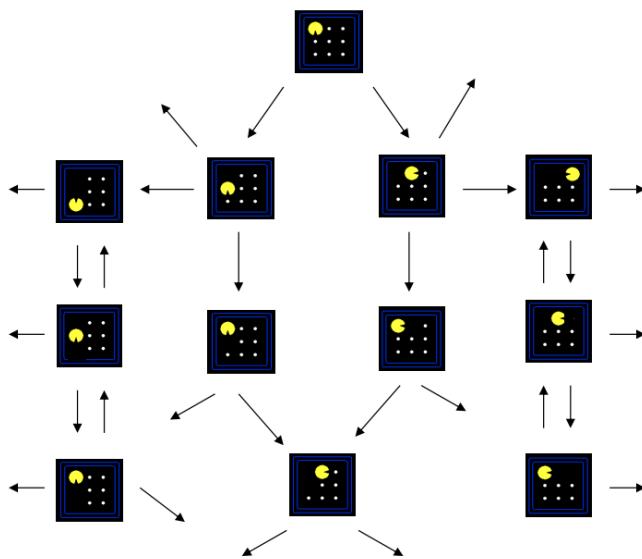


FIGURE 2.2 – Graph d'état partiel pour le jeu Pacman

Remarque: Il est fortement possible de ne pas pouvoir représenter un problème de recherche par un graphe d'espace d'état car il y a **trop d'états** ou que les états sont **continus**.

2.3 Arbres de Recherches

Définition 2.5. Arbre de Recherche

Un arbre de recherche est un arbre qui représente les états et les actions possibles. C'est la représentation d'un *Et si je prenais cette action ?* sur une certaine configuration de l'environnement.

- **Noeuds** : les états, plans pour arriver à ces états
- **Arêtes** : les actions
- **Enfants** : les états suivants (*successeurs*)

L'état initial est le noeud racine et l'état final est un noeud (ou plusieurs?). Les noeuds peuvent être représentés plusieurs fois, (**il est donc plus grand qu'un graphe d'espace d'état.**)

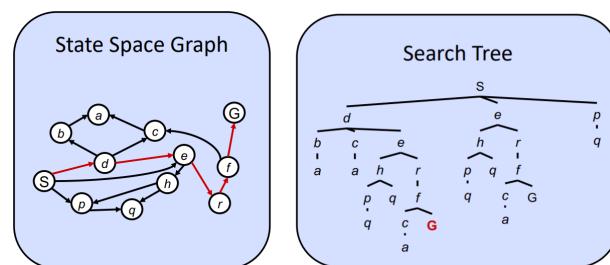


FIGURE 2.3 – Arbre de recherche vs Graphe d'espace d'état



Chaque noeuds dans un arbre de recherche, est un chemin dans le graphe d'espace d'état.

Note:-

L'arbre de recherche est construit au **fur et à mesure** de la recherche. En général, on essaye de minimiser sa taille.

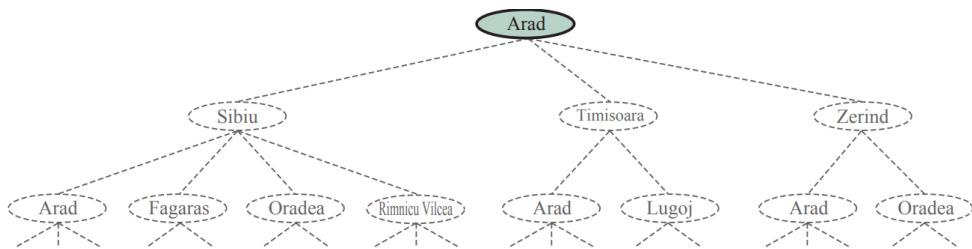


FIGURE 2.4 – Arbre de recherche de la figure 2.1

2.3.1 Recherche dans un arbre de recherche

Recherche dans un Arbre 1 Algorithme de recherche

```

function TREE-SEARCH(problème, stratégie)
    initialise un noeud avec l'état initial du problème
    loop
        if il ne peut plus y avoir d'état à explorer then
            return Erreur
        end if
        choisis un noeud non exploré selon la stratégie
        if le noeud est l'état final then
            return le plan qui mène à l'état final
        else
            Développe le noeud et ajoute ses enfants à l'arbre
        end if
    end loop
end function
  
```

Remarques:

1. La frontière est l'ensemble des noeuds construits non explorés de l'arbre
2. Pour développer un noeud de la frontière, on le retire de la frontière et on l'ajoute à l'ensemble des noeuds explorés ses enfants sont ajoutés à la frontière
3. La recherche dans un graphes est similaire à la recherche dans un arbre sauf qu'on doit vérifier si un noeud a déjà été visité avant de l'ajouter à la frontière

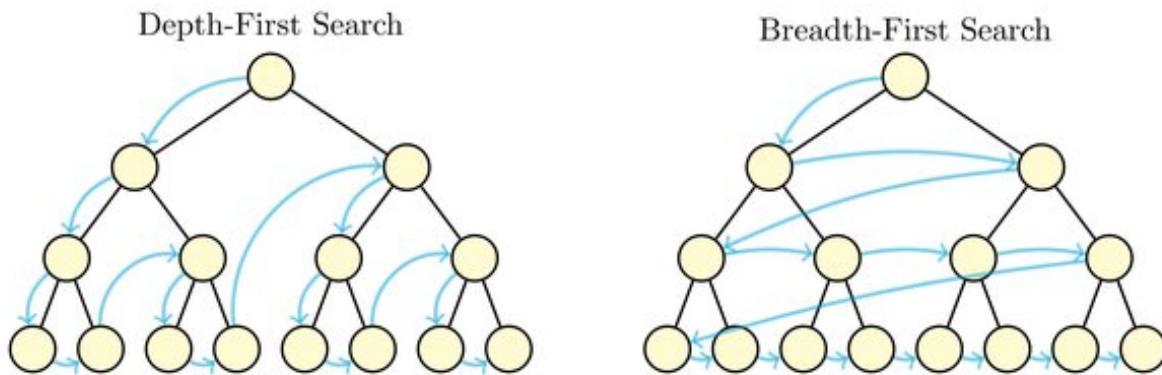


FIGURE 2.5 – Sens d'exécution BFS et DFS

2.4 Depth-First Search (DFS)

Définition 2.6. DFS

La recherche en profondeur (**DFS**) est une stratégie de recherche qui explore l'arbre en allant le plus loin possible dans une branche avant de revenir en arrière.

- **Frontière** : une pile (LIFO)
- **Stratégie** : on choisit le noeud le plus profond de la frontière et on l'étend

2.5 Breadth-First Search (BFS)

Définition 2.7. BFS

La recherche en largeur (**BFS**) est une stratégie de recherche qui explore l'arbre en allant le plus large possible dans une branche avant de revenir en arrière.

- **Frontière** : une file (FIFO)
- **Stratégie** : on choisit le noeud le moins profond de la frontière et on l'étend

DFS est meilleur que **BFS** dans les cas suivant :

- Si il y a des limitations de mémoire

BFS est meilleur que **DFS** dans les cas suivant :

- Si on veut trouver la solution la plus courte

2.6 Iterative deepening

Note:-

L'idée est d'avoir les avantages mémoire de **DFS** et la solution optimale de **BFS**

Définition 2.8. Iterative deepening

L'exploration itérative en profondeur (**Iterative deepening**) est une stratégie de recherche qui explore l'arbre en faisant une recherche en profondeur avec une limite de profondeur de 1, puis 2, puis 3, etc. jusqu'à ce que la solution soit trouvée.

Remarque: Même si cette algorithme visite plusieurs fois les mêmes noeuds, ça n'a pas vraiment d'impact car le nombre de noeuds est réduit. En effet, on mise de le trouver avant d'atteindre la limite de profondeur. Plus on avance dans l'arbre, plus le nombre de noeuds augmente de b (branching factor), nous misons donc sur le fait que la solution se trouve dans les premiers noeuds.

2.7 UCS

Définition 2.9. Uniform Cost Search

La recherche par coût uniforme (**UCS**) est une stratégie de recherche qui explore l'arbre en allant le plus loin possible dans une branche avant de revenir en arrière.

- **Frontière** : une file de priorité
- **Stratégie** : on choisit le noeud ayant le plus petit coût de la frontière

Pour analyser un algorithme, on va utiliser ces différentes propriétés :

- **Complet** : l'algorithme trouve toujours une solution si elle existe
- **Optimal** : l'algorithme trouve toujours la solution optimale (avec le plus petit coût)
- **Complexité en temps** : Combien de temps l'algorithme prend pour trouver une solution
- **Complexité en espace** : Combien de mémoire l'algorithme prend pour trouver une solution

TABLE 2 – Comparaison stratégie d'exploration

Critère	Largeur	Cout uniforme	Profondeur	Profondeur itérative
Complet	Oui	Oui	Non (cycle, ∞ noeuds)	Oui
Optimal	Oui	Oui	Non	Oui
Temps	$O(b^d)$	$O(b^{\frac{C^*}{\epsilon}})$	$O(b^m)$	$O(b^d)$
Espace	$O(b^d)$	$O(b^{\frac{C^*}{\epsilon}})$	$O(bm)$	$O(bd)$

3 Recherche Informée

Définition 3.1. Recherche Informée

La recherche **informée** est une stratégie de recherche qui utilise **des informations** sur l'état de l'environnement afin de le **guider** pour trouver une solution. Elle explore l'espace de recherche de manière **intelligente** grâce à des **heuristiques**. Ces heuristiques permettent de **quantifier la qualité** d'un état et donc de guider la recherche vers les états les plus prometteurs.

Remarque: La performance de ces algorithmes dépendent de la qualité de l'heuristique utilisée.

Définition 3.2. Heuristique

Une heuristique est une fonction qui permet d'estimer à quel point un état est **proche** de l'état final. Elles sont designées pour des problèmes spécifiques.

Exemples:

1. **Distance Euclidéenne** : la distance entre deux points en ligne droite
2. **Distance de Manhattan** : la distance entre deux points en ligne droite mais en ne pouvant se déplacer que sur les axes (pas en diagonale)

3.1 Greedy Best-First Search

Définition 3.3. Greedy Best-First Search

La recherche gloutonne (**Greedy Best-First Search**) est une stratégie de recherche qui explore l'arbre en choisissant le noeud qui semble le plus prometteur.

- **Frontière** : une file de priorité
- **Stratégie** : on choisit le noeud ayant la meilleure heuristique de la frontière (en fonction de la stratégie)

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

FIGURE 3.1 – Tableau de valeur d'heuristique

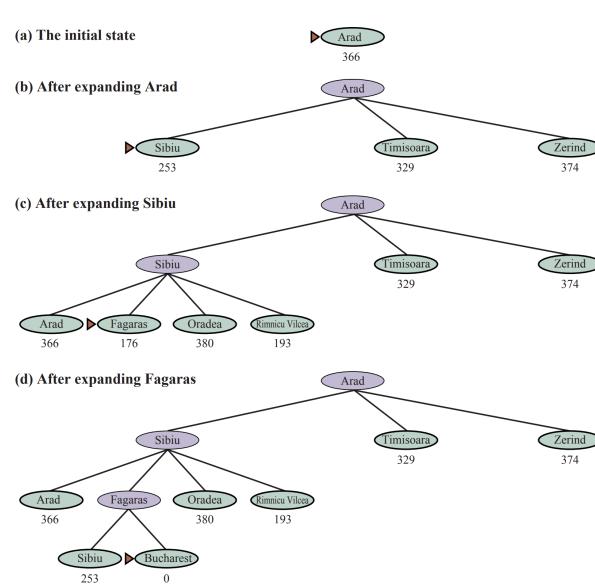


FIGURE 3.2 – Exécution de l'algorithme greedy

Remarques:

1. **Complet** : Non, peut aller dans des profondeurs infinies
2. **Optimal** : Non
3. **Complexité en temps** : $O(b^m)$
4. **Complexité en espace** : $O(b^m)$
5. Il est en général meilleur que **DFS**

3.2 A*

Définition 3.4. A*

L'algorithme **A*** est une stratégie de recherche qui explore l'arbre en choisissant le noeud qui semble le plus prometteur.

- **Frontière** : une file de priorité
- **Stratégie** : on choisit le noeud ayant le plus petit coût de la frontière

La fonction d'évaluation est la suivante : $f(n) = g(n) + h(n)$ où $g(n)$ est le coût pour atteindre le noeud n depuis la racine et $h(n)$ est le cout du noeud n pour aller jusqu'au but (*heuristique*).

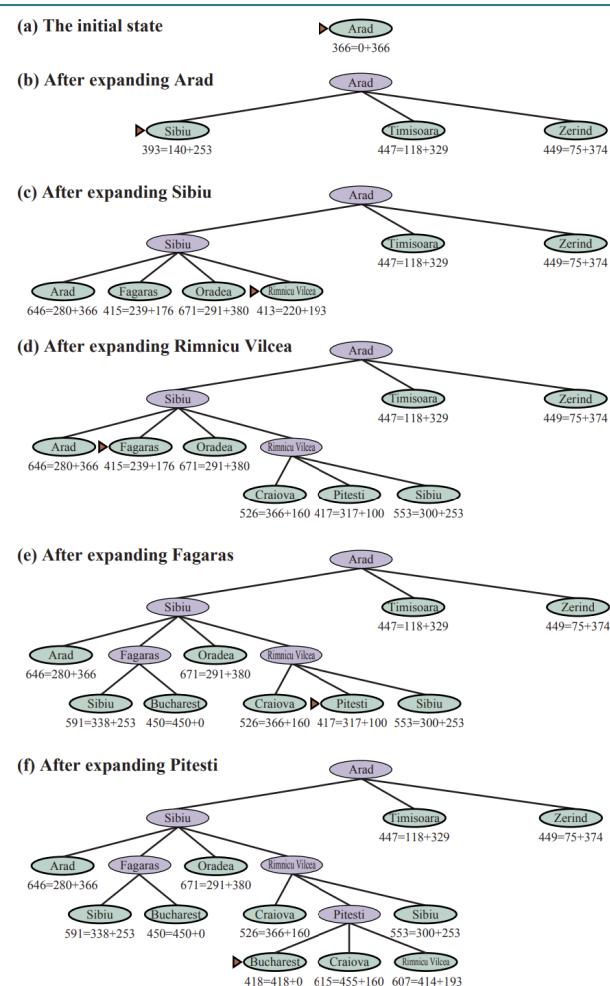


FIGURE 3.3 – Exécution de l'algo A*

Remarques:

1. **Complet** : Oui
2. **Optimal** : Oui si $h(n)$ est admissible
3. **Complexité en temps** : $O(b^d)$
4. **Complexité en espace** : $O(b^d)$

3.3 Creer des Heuristiques admissibles

Définition 3.5. Admissible

Une heuristique est admissible si elle ne surestime jamais le coût pour atteindre l'état final. $h(n) \leq h^*(n)$ où $h^*(n)$ est le coût réel pour atteindre l'état final. C'est une fonction **optimiste**

Définition 3.6. Cohérent

Une **heuristique** est **cohérente** si, pour chaque noeud n et un successeur n' de n généré par une action a , le coût estimé pour atteindre l'état final depuis n est inférieur ou égal au coût de l'action a plus le coût estimé pour atteindre l'état final depuis n' . $h(n) \leq c(n, a, n') + h(n')$ où $c(n, a, n')$ est le coût de l'action a pour aller de n à n' . Sinon, ça veut dire que l'heuristique surestime le coût pour atteindre l'état final.

Remarque: Une heuristique cohérente est toujours admissible.

Note:-

Lorsqu'on compare 2 heuristiques, on peut dire que l'heuristique h_1 est meilleure que h_2 si $h_1(n) \leq h_2(n) \forall n$

Afin de créer une heuristique admissible, on peut utiliser les méthodes suivantes :

- **Relaxation** : on simplifie le problème en enlevant des contraintes
- **Décomposition** : on décompose le problème en sous-problèmes qui sont plus facile à résoudre
- **Combinaison** : on combine plusieurs heuristiques

Il est aussi possible de stocker les valeurs réelles dans une **base de donnée** et de les utiliser pour calculer l'heuristique (qui est dcp dans la bdd).

4 Recherche Locale

à faire

5 Recherche Adversarial

Dans les chapitres précédents, nous avons vu comment résoudre des problèmes de recherche. Dans ce chapitre, nous allons nous concentrer sur les problèmes où nous devons **battre nos adversaires** dans des jeux à deux ou plusieurs joueurs.

5.1 Définition d'un jeu

Définition 5.1. Jeu

Un jeu est un problème de recherche avec les caractéristiques suivantes :

- **État initial**, s_0 : la position initiale du jeu.
- **Joueur**, $Player(s)$: le joueur qui doit jouer.
- **Actions**, $Action(s)$: les coups possibles pour un joueur.
- **Modèle de transition**, $Result(a, s)$: L'état s' qui résulte de l'action a dans l'état s
- **Fonction de terminal**, $isTerminal(s)$: la fonction qui détermine si le jeu est terminé
- **Utilité**, $Utility(s, p)$: la fonction qui détermine le score du jeu sur les états terminaux. Qui gagne, et combien. p est le joueur.

Note:-

Il y a plusieurs types de jeux :

- **Jeux à somme nulle** : la somme des utilités des joueurs est toujours égale à 0. Si l'un gagne, l'autre perd.
- **Jeux à somme général** : Les agents peuvent avoir des utilités différentes. Si l'un gagne, l'autre ne perd pas forcément. (*Coop, ...*)
- **Jeux d'équipes** : Les agents jouent en équipe contre d'autres agents.

Les algorithmes de recherche que nous avons vu précédemment ne sont pas adaptés pour les jeux car ils ne prennent pas en compte le fait que l'adversaire joue aussi. Nous allons donc voir des algorithmes qui vont prendre en compte le fait que l'adversaire joue aussi et vont alors recommander le meilleur coup à jouer selon toutes les possibilités.

5.2 Minimax

Définition 5.2. Minimax

L'algorithme **Minimax** est un algorithme qui permet de trouver le meilleur coup à jouer dans un jeu **déterministe à somme nulle**. Le principe est simple, nous imaginons 2 joueurs qui jouent l'un contre l'autre, **Min** et **Max**. Le but de **Max** est de maximiser l'utilité et le but de **Min** est de minimiser l'utilité sachant que l'autre joueur va jouer de manière optimale.

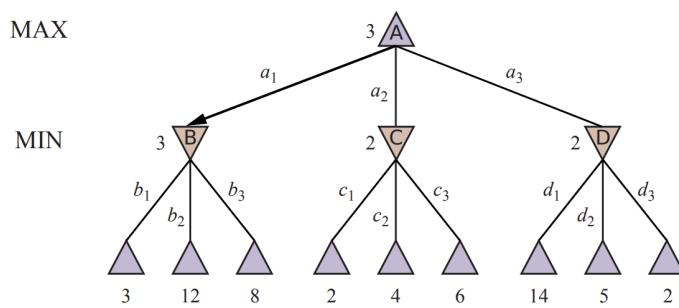


FIGURE 5.1 – Minimax dans un arbre.

L'algorithme va généré l'arbre de recherche jusqu'aux étas **terminaux** pour utiliser la fonction d'utilité. Il traite ensuite ces valeurs en remontant l'arbre et les choisit en fonction du joueur qui doit jouer (**Min** ou **Max**)

• **Note:-**

Pour ne pas générer tous l'arbre, on peut utiliser une **profondeur limitée**.

Si il y a plus de 2 joueurs, nous pouvons assigner à chaque noeud un tuple de valeurs qui représente l'utilité pour chaque joueur. Chaque joueur va alors choisir le coup qui maximise son utilité.

Remarque: Cet algorithme est basé sur **DFS**.

- **Complexité en temps** : $O(b^m)$
- **Complexité en espace** : $O(bm)$

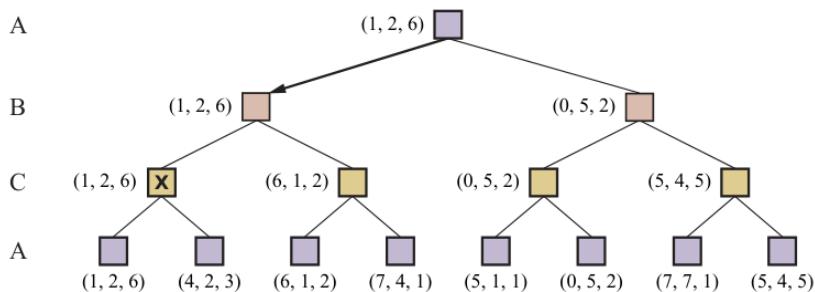


FIGURE 5.2 – Représentation en Multijoueur

Cette algorithme n'est pas adapté pour les jeux où il y a beaucoup de noeuds car il va générer tout l'arbre de recherche. Nous allons donc voir des algorithmes qui vont couper les branches qui ne sont pas intéressantes.

Minimax 2 Algorithme Minimax

```

function MINIMAX-SEARCH(game, state)
    player ← game.To-MOVE(state)
    if player = max then
        value, action ← MAX-VALUE(game, state)
    else
        value, action ← MIN-VALUE(game, state)
    end if
    return action
end function

function MAX-VALUE(game, state)
    if game.TERMINAL-TEST( state) then
        return game.UTILITY(state, player), null
    end if
    value, action ← −∞, null
    for action in game.ACTIONS(state) do
        value2, action2 ← MIN-VALUE(game, game.RESULT(state, action))
        if value2 > value then
            value, action ← value2, action2
        end if
    end for
    return value, action
end function

function MIN-VALUE(game, state)
    if game.TERMINAL-TEST( state) then
        return game.UTILITY(state, player), null
    end if
    value, action ← ∞, null
    for action in game.ACTIONS(state) do
        value2, action2 ← MAX-VALUE(game, game.RESULT(state, action))
        if value2 < value then
            value, action ← value2, action2
        end if
    end for
    return value, action
end function

```

5.3 Alpha-Beta pruning

Définition 5.3. Alpha-Beta pruning

L'algorithme **Alpha-Beta pruning** est un algorithme qui permet de trouver le meilleur coup à jouer dans un jeu **déterministe à somme nulle**. Il est basé sur l'algorithme **Minimax** mais il va **couper** les branches qui ne sont pas intéressantes. Il utilise 2 paramètres, α et β qui représentent les valeurs minimales et maximales que l'on a trouvées jusqu'à présent.

- α est la meilleure option (valeur la plus haute) que le joueur **Max** est assuré d'obtenir.
Au MOINS.
- β est la meilleure option (valeur la plus basse) que le joueur **Min** est assuré d'obtenir.
Au PLUS.

Si le noeud est un noeud **Max**, on va mettre à jour α avec la valeur maximale trouvée jusqu'à présent. Si le noeud est un noeud **Min**, on va mettre à jour β avec la valeur minimale trouvée jusqu'à présent. Tous les noeuds qui sont entre α et β ne seront pas visités car ils ne sont pas intéressants.

Remarque: Le meilleur situation pour cette algorithme est quand les meilleurs coups sont toujours le premier coup à être évalué (*plus à gauche*)

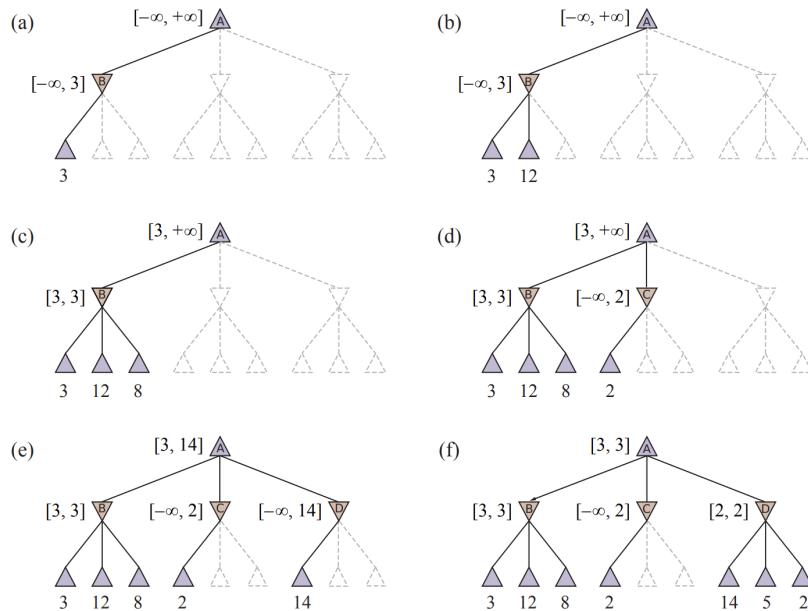


FIGURE 5.3 – Alpha-Beta pruning dans un arbre

Remarque: La complexité en temps est généralement plus petite que celle de minimax

- **Complexité en temps** : $O(b^{m/2})$ avec les noeuds dans le **bon** ordre et $O(b^{3m/4})$ si les noeuds sont visités aléatoirement.
- **Complexité en espace** : $O(bm)$ et $O(\sqrt{b})$ si les noeuds sont dans le bon ordre.

Note:-

Même si l'on ne visite pas tous les noeuds, on a quand même trouver la solution optimale.

Si cet algorithme n'est pas suffisant, on peut utiliser une fonction d'évaluation qui permet d'évaluer les noeuds qui ne sont pas des états terminaux pour estimer leur utilité.

$\alpha - \beta$ pruning 3 Algorithme $\alpha - \beta$ pruning

```

function ALPHA-BETA-SEARCH(game, state)
    player ← game.TO-MOVE(state)
    value, action ← MAX-VALUE(game, state,  $-\infty$ ,  $+\infty$ )
    return action
end function

function MAX-VALUE(game, state,  $\alpha$ ,  $\beta$ )
    if game.TERMINAL-TEST( state) then
        return game.UTILITY(state, player), null
    end if
    value, action ←  $-\infty$ , null
    for action in game.ACTIONS(state) do
        value2, action2 ← MIN-VALUE(game, game.RESULT(state, action),  $\alpha$ ,  $\beta$ )
        if value2 > value then
            value, action ← value2, action2
             $\alpha$  ← MAX( $\alpha$ , value)
        end if
        if value  $\geq \beta$  then
            return value, action
        end if
    end for
    return value, action
end function

function MIN-VALUE(game, state,  $\alpha$ ,  $\beta$ )
    if game.TERMINAL-TEST( state) then
        return game.UTILITY(state, player), null
    end if
    value, action ←  $\infty$ , null
    for action in game.ACTIONS(state) do
        value2, action2 ← MAX-VALUE(game, game.RESULT(state, action),  $\alpha$ ,  $\beta$ )
        if value2 < value then
            value, action ← value2, action2
             $\beta$  ← MIN( $\beta$ , value)
        end if
        if value  $\leq \alpha$  then
            return value, action
        end if
    end for
    return value, action
end function

```

5.4 Expectimax

Définition 5.4. Expectimax

L'algorithme Expectimax est une technique d'exploration d'arbre de décision utilisée pour la prise de décision dans des environnements incertains ou probabilistes. Il est souvent appliqué dans des jeux et des situations où les actions des adversaires ou des événements aléatoires peuvent influencer le déroulement du jeu.

C'est donc une variante de l'algorithme **Minimax** qui va prendre en compte les noeuds de **chance**. Ces noeuds de chance sont des noeuds où le résultat dépend du hasard ou de l'incertitude. La valeur d'un noeud de chance est la moyenne pondérée des valeurs de ses fils.

Remarque: Les noeuds de chance sont comme les noeuds du joueur **MIN** mais cette fois-ci, le résultat n'est pas **certain** (Poker, lancé de dés, mauvais move, ...).

• Note:-

Alors que minimax représente le **pire** des cas pour un joueur (l'adversaire joue de manière optimale), expectimax représente le **cas moyen** pour un joueur (l'adversaire peut aussi bien jouer de manière optimale que de manière "aléatoire"). Expectimax permet donc de prendre des opportunités que minimax ne prendrait pas car il est trop pessimiste, restrictif.

Expectimax 4 Algorithme Expectimax

```

function EXPECTIMAX-SEARCH(game, state)
    player ← game.To-MOVE(state)
    if player = max then
        value, action ← MAX-VALUE(game, state)
    else
        value, action ← EXP-VALUE(game, state)
    end if
    return action
end function

function MAX-VALUE(game, state)
    if game.TERMINAL-TEST( state) then
        return game.UTILITY(state, player), null
    end if
    value, action ←  $-\infty$ , null
    for action in game.ACTIONS(state) do
        value2, action2 ← EXP-VALUE(game, game.RESULT(state, action))
        if value2 > value then
            value, action ← value2, action2
        end if
    end for
    return value, action
end function

function EXP-VALUE(game, state)
    if game.TERMINAL-TEST( state) then
        return game.UTILITY(state, player), null
    end if
    value, action ←  $\infty$ , null
    for action in game.ACTIONS(state) do
        p ← game.PROBABILITY(state, action)
        value2, action2 ← MAX-VALUE(game, game.RESULT(state, action))
        value ← value + p * value2
    end for
    return value, action
end function

```

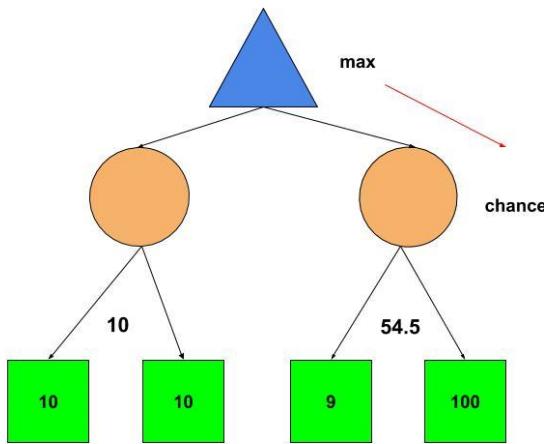


FIGURE 5.4 – Arbre Expectimax

Note:-

Il n'est cependant pas possible d'élaguer l'arbre de recherche car pour évaluer un noeud de chance, il faut évaluer tous ses fils.

5.5 Monte Carlo Tree Search (MCTS)

Définition 5.5. MCTS

L'algorithme **Monte Carlo Tree Search** est un algorithme de recherche d'arbre qui utilise des simulations "aléatoires" ou selon une certaine politique pour trouver la meilleure action à jouer. En effet, plus on fait de simulations, plus on se rapproche de la vraie valeur de l'état. Sachant cela, on va donc faire des simulations sur les noeuds qui sont les plus intéressants en fonction du nombre de simulation à partir de ce noeud et de la valeur de ce noeud (**utilité**, **wins**). Il y a 4 étapes dans cet algorithme :

1. **Sélection** : L'algorithme commence par choisir un chemin dans l'arbre existant en utilisant une stratégie qui combine l'exploration des noeuds prometteurs et l'exploitation des noeuds déjà visités.
2. **Expansion** : Une fois qu'un noeud feuille est atteint, l'algorithme ajoute de nouveaux noeuds pour représenter les actions possibles à partir de cet état.
3. **Simulation** : Des simulations aléatoires sont effectuées à partir des noeuds nouvellement créés (tous?) (ou des noeuds déjà existants) pour estimer la valeur de chaque action. On simule jusqu'à atteindre un état final ou une condition d'arrêt.
4. **Backpropagation** : Les résultats des simulations sont propagés vers le haut de l'arbre, mettant à jour les statistiques des noeuds visités pour refléter les nouvelles informations obtenues.

Pour choisir le noeud à explorer, on utilise l'équation suivante :

$$UCB(i) = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}} \quad (1)$$

Où w_i est le nombre de victoires lors des simulations à partir du noeud i , n_i est le nombre de simulations à partir du noeud i , N est le nombre de simulations total du parent de i et c est un

paramètre d'exploration qui contrôle l'importance de l'exploration par rapport à l'exploitation. Plus c est grand, plus l'exploration est importante.

A la fin, on choisit le noeud qui a eu le plus de simulations. En effet, s'il a eu beaucoup de simulations, cela veut dire qu'il est intéressant de le choisir.

• **Note:-**

Plus le nombre de simulation se rapproche de ∞ , plus il se rapprochera du choix optimal.

6 Probabilités

Les raisonnements probabilistes en **Intelligence Artificielle** sont basés sur la théorie des probabilités et permettent de modéliser des situations où l'on ne peut pas prédire avec certitude le résultat d'une action (*environnements non-déterministes*).

6.1 Quelques définitions

Définition 6.1. Événement élémentaire

Un état possible de l'environnement. Il est souvent noté ω .

Définition 6.2. Univers

L'ensemble de tous les événements élémentaires possibles. Il est souvent noté Ω .

Définition 6.3. Variable aléatoire

Une **variable aléatoire** est une fonction qui associe à chaque événement élémentaire d'un espace probabilisé un nombre réel. On note A une variable aléatoire et ω une valeur prise par A . On note $P(A = \omega)$ la probabilité que A prenne la valeur ω . On note $P(A)$ la loi de probabilité de A .

Remarque: C'est une manière de quantifier de numériquement les résultats d'une expérience aléatoire.

Fonction qui prend en entrée un événement élémentaire et qui renvoie un nombre réel (quantification).

6.2 Rappel proba

Définition 6.4. Probabilité Conjointes

La probabilité conjointe de deux variables aléatoires A et B est la probabilité de l'événement où A prend la valeur x **ET** B prend la valeur y .

$$\begin{aligned} P(A = x, B = y) &= P(A = x \cap B = y) \\ &= P(A = x | B = y)P(B = y) \\ &= P(B = y | A = x)P(A = x) \end{aligned}$$

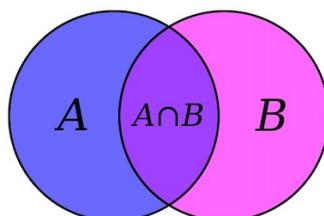


FIGURE 6.1 – Diagramme de Venn de $P(A \cap B)$

Définition 6.5. Probabilité d'une disjonction

La probabilité d'une disjonction de deux variables aléatoires A et B est la probabilité de l'événement où A prend la valeur x **OU** B prend la valeur y .

$$P(A = x \cup B = y) = P(A = x) + P(B = y) - P(A = x \cap B = y) \quad (2)$$

La soustraction est nécessaire pour éviter de compter deux fois la probabilité de l'intersection.

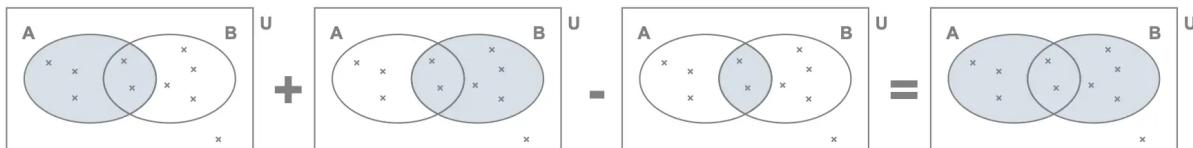


FIGURE 6.2 – Diagramme de Venn de $P(A \cup B)$

Définition 6.6. Probabilité Marginale

La probabilité marginale d'une variable aléatoire A est la probabilité de l'événement où A prend la valeur x .

$$P(A = x) = \sum_{y \in B} P(A = x \cap B = y) \quad (3)$$

Où B prend toutes ses valeurs possibles.

Remarque: C'est la probabilité sur un sous-ensemble de variables aléatoires.

Définition 6.7. Distribution de probabilité

Une distribution de probabilité est une fonction qui associe à chaque événement élémentaire d'un espace probabilisé un nombre réel positif. La somme de toutes les probabilités doit être égale à 1.

$$\sum_{\omega \in \Omega} P(\omega) = 1 \quad (4)$$

Définition 6.8. Probabilité Conditionnelle

La probabilité conditionnelle de deux variables aléatoires A et B est la probabilité de l'événement où A prend la valeur x sachant que B prend la valeur y .

$$P(A = x | B = y) = \frac{P(A = x \cap B = y)}{P(B = y)} \quad (5)$$

Une probabilité conditionnelle peut être vue comme une probabilité **renomalisée** afin de respecter la contrainte de somme à 1. En effet, le facteur $P(B = y)$ est égale à la probabilité marginale $\alpha = \sum_{x \in A} P(A = x \cap B = y)$. Pour renormaliser une probabilité/distribution, il suffit de diviser chaque probabilité par la somme de toutes les probabilités. Cette constante de normalisation est appelée **facteur de normalisation** et est notée α .

Exemple: Prenons ce tableau de probabilité conjointe $P(T, W)$

T	W	P
hot	sun	0.4
hot	rain	0.1
cold	sun	0.2
cold	rain	0.3

TABLE 3 – Tableau $P(T \cap W)$

La somme de toutes les probabilités est égale à 1. Ce qui est normal car c'est une distribution de probabilité. Si nous voulons trouver la probabilité conditionnelle $P(W|T = \text{cold})$, on procède comme suit :

- On sélectionne les lignes où $T = \text{cold}$. (comme dans une bdd)

T	W	P
cold	sun	0.2
cold	rain	0.3

TABLE 4 – Tableau $P(T = \text{cold} \cap W)$

Nous remarquons que la somme des probabilités n'est plus égale à 1.

- On renormalise les probabilités en divisant chaque probabilité par la somme de toutes les probabilités.

$$\alpha = \frac{1}{\sum_{x \in W} P(T = \text{cold} \cap W = x)} = \frac{1}{0.2 + 0.3} = \frac{1}{0.5}$$

T	W	P
cold	sun	$0.4 = 0.2 \cdot \alpha$
cold	rain	$0.6 = 0.3 \cdot \alpha$

TABLE 5 – Tableau $P(W|T = \text{cold})$ renormalisé

Nous remarquons que la somme des probabilités est de nouveau égale à 1.

Remarque: Une distribution de probabilité est une variable aléatoire. On peut donc parler de probabilité conjointe, marginale, conditionnelle, etc. sur une distribution de probabilité.

Théorème 6.1. Bayes

Ce théorème permet de calculer une probabilité conditionnelle à partir de la probabilité conditionnelle inverse. *inversement du conditionnement*

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)} \quad (6)$$

$P(A)$ est appelé la probabilité **a priori** de A . $P(A|B)$ est appelé la probabilité **a posteriori** de A . A posteriori = après avoir observé B .

Théorème 6.2. Multiplication

Ce théorème permet de calculer une probabilité conjointe à partir de probabilités conditionnelles.

$$P(A \cap B) = P(A|B)P(B) \Leftrightarrow P(A|B) = \frac{P(A \cap B)}{P(B)} \quad (7)$$

Théorème 6.3. Chainage

Ce théorème permet de calculer une probabilité conjointe à partir de probabilités conditionnelles.

$$P(A_1 \cap A_2 \cap \dots \cap A_n) = P(A_1)P(A_2|A_1)P(A_3|A_1 \cap A_2)\dots P(A_n|A_1 \cap A_2 \cap \dots \cap A_{n-1}) \quad (8)$$

$$= \prod_{i=1}^n P(A_i | \bigcap_{j=1}^{i-1} A_j) \quad (9)$$

Ce théorème est une généralisation du théorème de multiplication.

Remarque: C'est la probabilité d'une variable aléatoire sachant que toutes les précédentes ont eu lieu.

6.3 Inférence

Définition 6.9. Inférence

L'inférence est le processus de déduction de nouvelles informations à partir d'informations déjà connues. En probabilité, c'est le processus de déduction de probabilités à partir de probabilités déjà connues. Comme par exemple, calculer la probabilité conditionnelle $P(A|B)$ (inconnue) à partir de la probabilité conjointe $P(A \cap B)$ (connue).

Remarque: En général, nous avons des probabilités conjointes et nous voulons calculer des probabilités conditionnelles.

6.3.1 Inférence par Énumération

Définition 6.10. Inférence par Énumération

L'inférence par énumération est une méthode d'inférence qui consiste à énumérer toutes les valeurs possibles des variables aléatoires et à calculer la probabilité de chaque valeur.

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{\sum_{x \in A} P(A = x \cap B)}{P(B)} \quad (10)$$

Cette méthode est très coûteuse en temps et en mémoire car il faut énumérer toutes les valeurs possibles. Il y a 3 types de variables :

- **Variables d'observation** : Variables connues E_i qui vont nous permettre de faire l'inférence
- **Variables cachées** : Variables inconnues qui ne nous intéressent pas H_i
- **Variable de Requête** : Variable inconnue qui nous intéresse Q_i

Toutes les variables sont des variables aléatoires X_i . Nous recherchons la probabilité conditionnelle $P(Q_i|E_1, \dots, E_n)$.

(pas trop compris en vrai)

Remarque: Cette méthode n'est pas utilisable en pratique car elle est trop coûteuse en temps et en mémoire. Complexité exponentielle en fonction du nombre de variables cachées.

6.4 Indépendance

Définition 6.11. Indépendance

Deux variables aléatoires A et B sont indépendantes si et seulement si

- $P(A|B) = P(A)$ (savoir B ne change pas la probabilité de A) ou
- $P(B|A) = P(B)$ (savoir A ne change pas la probabilité de B) ou
- $P(A \cap B) = P(A)P(B)$ (il n'y a pas d'impact mutuel)

Elle permet de simplifier les calculs de probabilités conjointes. Si deux variables aléatoires sont indépendantes, on note $A \perp B$ (avec 2 barres).

Définition 6.12. Indépendance Conditionnelle

Deux variables aléatoires A et B sont indépendantes conditionnellement sachant une variable aléatoire C si et seulement si

- $P(A|B \cap C) = P(A|C)$ ou
- $P(A \cap B|C) = P(A|C)P(B|C)$

Dans ce cas, A et B sont indépendantes sachant C . Cela signifie que si on connaît la valeur de C , savoir la valeur de B ne change pas la probabilité de A . Noté $A \perp B|C$ (avec 2 barres).

Remarque: Si il n'y avait pas de condition et que A et B sont indépendantes, on aurait $P(A|B) = P(A)$. Sauf que maintenant on a une condition C qui est réalisée, et elle ne peut être supprimée

7 Réseaux Bayesiens

Définition 7.1. Réseaux Bayésiens

Un réseau bayésien est un graphe orienté acyclique (DAG) dont les nœuds représentent des variables aléatoires et les arcs représentent des dépendances conditionnelles.

Ils permettent de représenter des distributions de probabilités conjointes de manière compacte, de construire des modèles de raisonnement probabiliste et de faire de l'inférence.

- Les noeuds représentent des variables aléatoires
- Les arcs représentent des dépendances (*causalités ?*) conditionnelles ainsi que des distributions de probabilités pour chaque variable aléatoire **étant donné** ses parents

Façon compacte de représenter des **probabilités conjointes**.

La topologie du **RB** modélise les relations d'influence entre les variables aléatoires.

Un arc $X \rightarrow Y$ signifie que X influence Y .

Si X n'a pas de parents, alors sa distribution de probabilité est dite **inconditionnelle** ou **à priori**. Si X a des parents, alors sa distribution de probabilité est dite **conditionnelle** ou **à posteriori**. Chaque nœuds est associé à une table de probabilités conditionnelles (CPT) qui spécifie la probabilité de chaque valeur de la variable aléatoire étant donné les valeurs de ses parents.

Exemple: Voici un exemple du livre *Artificial Intelligence : A Modern Approach* de Stuart Russel et Peter Norvig. Considérons la situation suivante :

- Je suis au travail et mes voisins Marie et John m'ont promis de **m'appeler** chaque fois que mon **alarme** se déclenche.
- **Jean m'appelle** pour me dire que mon alarme s'est déclenchée.
 - Cependant, il **la confond** parfois avec la sonnerie du téléphone
- **Marie m'appelle pas toujours**
 - Elle écoute de la musique et ne l'entend pas toujours
 - Mon alarme peut également sonner à cause de **séismes**.
 - → **Comment conclure qu'il y a un cambriolage ?**

On représente cette situation par le réseau bayésien suivant :

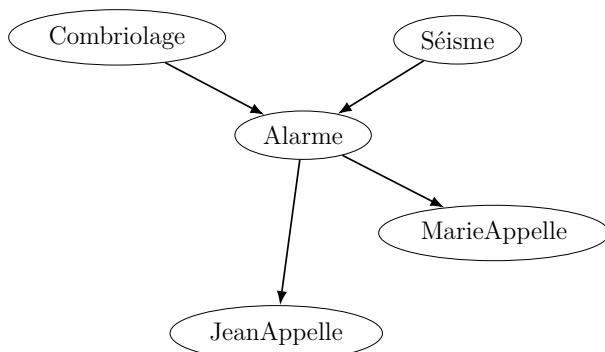


FIGURE 7.1 – Réseau bayésien de l'exemple

Voici les probabilités conditionnelles associées à ce réseau bayésien :

$P(Combriolage)$	$P(\neg Combriolage)$
0.001	0.999

FIGURE 7.2 – Probabilités de *Combriolage*

$P(Séisme)$	$P(\neg Séisme)$
0.002	0.998

FIGURE 7.3 – Table des probabilités de *Séisme*

<i>Combriolage</i>	<i>Séisme</i>	$P(Alarme)$
<i>V</i>	<i>V</i>	0.95
<i>V</i>	<i>F</i>	0.94
<i>F</i>	<i>V</i>	0.29
<i>F</i>	<i>F</i>	0.001

FIGURE 7.4 – Table des probabilités conditionnelles de *Alarme*

<i>Alarme</i>	$P(JeanAppelle)$
<i>V</i>	0.90
<i>F</i>	0.05

FIGURE 7.5 – Table des probabilités conditionnelles de *JeanAppelle*

<i>Alarme</i>	$P(MarieAppelle)$
<i>V</i>	0.70
<i>F</i>	0.01

FIGURE 7.6 – Table des probabilités conditionnelles de *MarieAppelle*

Note:-

Les réseaux bayésiens peuvent avoir des variables aléatoires continues ou discrètes.

Les arcs entre les noeuds peuvent représenter une relation de causalité comme par exemple. *Rain* → *Traffic* signifie que Rain cause Traffic. Cependant, ce n'est pas toujours le cas. De manière générale, les arcs représentent des dépendances conditionnelles. Cela signifie que si on connaît la valeur des parents d'un noeuds, alors on peut déduire la valeur du noeud et le reste n'a pas d'importance.

Nous savons que par définition, $P(A, B) = P(A|B)P(B)$. Nous pouvons donc écrire la probabilité conjointe d'un réseau bayésien comme suit :

$$P(X_1, X_2, \dots, X_n) = \prod_{i=1}^n P(X_i | Parents(X_i)) \quad (11)$$

Où $Parents(X_i)$ est l'ensemble des parents directs de X_i dans le réseau bayésien.

Exemple: En utilisant le réseau bayésien de l'exemple précédent, nous pouvons calculer la probabilité conjointe de toutes les variables aléatoires comme suit :

$$\begin{aligned} P(C = F, S = F, A = V, J = V, M = V) &= P(C = F)P(S = F)P(A = V | C = F, S = F) \\ &\quad P(J = V | A = V)P(M = V | A = V) \\ &= 0.999 \times 0.998 \times 0.001 \times 0.90 \times 0.70 \\ &= 0.000628 \end{aligned}$$

Pour calculer les probabilités marginales, on peut ignorer les noeuds **dont les descendants ne sont pas les noeuds observés**

Exemple:

$$\begin{aligned} P(C = F \cap A = V) &= \sum_s \sum_j \sum_m P(C = F, S = s, A = V, J = j, M = m) \\ &= \sum_s P(A = V | C = f, s)P(C = F)P(S = s) \end{aligned}$$

On peut ignorer J et M car ils ne sont pas des descendants de C qui est observé. Cependant, on ne peut pas ignorer S car A est un descendant de S et A est observé.

7.1 Inférence

7.1.1 Inférence par énumération

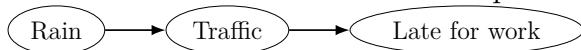
Remarque: N'importe quelles probabilités peuvent être utilisées à partir de la distribution de probabilités conjointes.

$$\begin{aligned} P(B|j, m) &= \alpha \sum_{e,a} P(B, e, a, j, m) \\ &= \alpha \sum_{e,a} P(B)P(e)P(a|B, e)P(j|a)P(m|a) \end{aligned}$$

Où α est une constante de normalisation. C'est comme si on marginalisait sur toutes les variables aléatoires sauf B qui est la variable aléatoire dont on veut calculer la probabilité. Le problème est que le nombre de calculs à effectuer est exponentiel en fonction du nombre de variables aléatoires.

Voici la procédure pour calculer une probabilité marginale avec l'inférence par énumération :

1. **Initialisation :** Toutes les tables de probabilités conditionnelles sont les facteurs initiaux.



Les probabilités conditionnelles associées à ce réseau bayésien :

$P(Rain)$	$P(\neg Rain)$
0.1	0.9

TABLE 6 – Tableau des probabilités de $P(R)$

$Rain$	$Traffic$	0.8
$Rain$	$\neg Traffic$	0.2
$\neg Rain$	$Traffic$	0.1
$\neg Rain$	$\neg Traffic$	0.9

TABLE 7 – Tableau des probabilités conditionnelles de $P(T|R)$

$Traffic$	$Late$	0.3
$Traffic$	$\neg Late$	0.7
$\neg Traffic$	$Late$	0.1
$\neg Traffic$	$\neg Late$	0.9

TABLE 8 – Tableau des probabilités conditionnelles de $P(L|T)$

Pour calculer $P(Late)$, on doit calculer la probabilité conjointe de toutes les variables aléatoires.

$$\begin{aligned} P(Late) &= \sum_{r,t} P(r, t, L) \\ &= \sum_{r,t} P(r)P(t|r)P(l|t) \end{aligned}$$

Remarque: Si *Late* avait la valeur observée *Vrai*, alors dans le tableau 8, on ne considère que les lignes où *Late* est vrai.

2. **Propagation/Join** : On propage les facteurs en multipliant les facteurs qui ont des variables en commun. C'est comme si on faisait une jointure sur les variables en commun. Même idée si on doit faire un join avec +2 variables. Par exemple, pour **join** *R*, on a $P(R) \times P(T|R) \Rightarrow P(R, T)$. la table de probabilités conditionnelles de $P(R, T)$ est :

<i>R</i>	<i>T</i>	0.08 = 0.1 · 0.8
<i>R</i>	$\neg T$	0.02 = 0.1 · 0.02
$\neg R$	<i>T</i>	0.09 = 0.9 · 0.1
$\neg R$	$\neg T$	0.81

TABLE 9 – Tableau des probabilités conditionnelles de $P(R, T)$

3. **Elimination** : On élimine les variables qui ne sont pas observées en sommant les facteurs qui ont les mêmes variables. Attention à ne pas supprimer les variables qui sont observées ou bien celle qui nous intéresse. On élimine jusqu'à ce qu'il nous reste que la variable dont on veut calculer la probabilité. La table 9 devient, lorsque on élimine *R* :

<i>T</i>	0.17
$\neg T$	0.83

TABLE 10 – Tableau des probabilités conditionnelles de $P(T)$

On a additionné les lignes où *T* avait la même valeur car nous voulons supprimer *R*.

7.1.1.1 Elimination de variables

L'idée est de réduire le nombre de calculs à effectuer en déplaçant les sommes vers l'intérieur de l'équation.

$$\begin{aligned} P(B|j, m) &= \alpha \sum_{e,a} P(B)P(e)P(a|B, e)P(j|a)P(m|a) \\ &= \alpha P(B) \sum_e P(e) \sum_a P(a|B, e)P(j|a)P(m|a) \end{aligned}$$

De cette manière on élimine les variables qui sont répétées dans les sommes.

L'idée est de marginaliser (éliminer) le plus tôt possible une variable (qui n'est pas utile).



Tu ne peux marginaliser une variable tant que tu n'as pas join les facteurs qui ont la variables en commun.

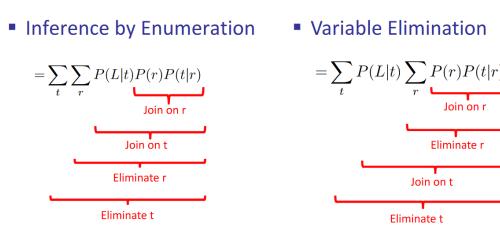


FIGURE 7.7 – Enumératin vs VE

Si il y a des variables observée. Lors des jointures, on ne considère que les lignes où les variables observées ont la valeur observée. Lorsqu'à la fin, il ne reste que la variable dont on veut calculer la probabilité ainsi que les variables observées, on normalise le résultat en divisant par la somme des probabilités.

E.g. for $P(L | +r)$, we would end up with:

$P(+r, L)$	Normalize	$P(L +r)$										
<table border="1"> <tr> <td>+r</td> <td>+l</td> <td>0.026</td> </tr> <tr> <td>+r</td> <td>-l</td> <td>0.074</td> </tr> </table>	+r	+l	0.026	+r	-l	0.074	→	<table border="1"> <tr> <td>+l</td> <td>0.26</td> </tr> <tr> <td>-l</td> <td>0.74</td> </tr> </table>	+l	0.26	-l	0.74
+r	+l	0.026										
+r	-l	0.074										
+l	0.26											
-l	0.74											

- **Query:** $P(Q|E_1 = e_1, \dots E_k = e_k)$
- **Start with initial factors:**
 - Local CPTs (but instantiated by evidence)
- **While there are still hidden variables (not Q or evidence):**
 - Pick a hidden variable H
 - Join all factors mentioning H
 - Eliminate (sum out) H
- **Join all remaining factors and normalize**

FIGURE 7.8 – VE avec evidence + algo

Remarque: $P(B|j, m) = \propto P(B, j, m)$ Cela signifie que l'on peut calculer la probabilité conditionnelle en calculant la probabilité conjointe. Il suffit ensuite de normaliser le résultat en divisant par la somme des probabilités.

7.2 D-Séparation

Définition 7.2. D-Séparation

La D-séparation dans un réseau bayésien est un outil pour déterminer l'indépendance conditionnelle entre des ensembles de variables en analysant les chemins entre ces variables dans le graphe du réseau bayésien. Elle est basée sur la notion de **triplet actif** et **triplet inactif** qui forment un chemin **actif** ou **inactif**.

- **Chemin** : Un chemin entre deux nœuds A et B dans un réseau bayésien est une séquence de liens qui connectent ces deux nœuds. Il est divisé en **triplets** et n'est pas dirigé.
- **Chemin Actif** : Un chemin est actif s'il n'est pas bloqué par des triplets inactifs. Il peut transmettre l'information entre les nœuds A et B.
- **Chemin Inactif** : Un chemin est inactif s'il contient un ou plusieurs triplets inactifs. Il ne peut pas transmettre l'information entre les nœuds A et B.

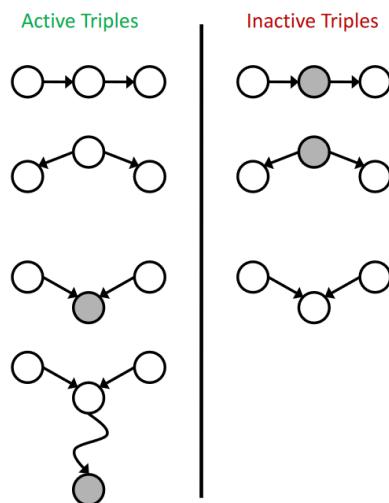


FIGURE 7.9 – Table pour savoir si un triplet est actif ou pas

Remarque: Cette technique est utile pour savoir si deux variables aléatoires sont indépendantes conditionnellement. Il n'est pas toujours facile, lors du design du réseau bayésien, de savoir quelles variables aléatoires sont indépendantes conditionnellement.

7.3 Echantillonage

7.3.1 Prior

Remarque: L'inférence par énumération est très coûteuse en temps de calcul. L'inférence par échantillonage est une alternative qui est plus rapide mais qui est moins précise. L'idée est de générer des échantillons à partir du réseau bayésien et de compter le nombre d'échantillons qui satisfont la condition. On peut ensuite calculer la probabilité en divisant le nombre d'échantillons qui satisfont la condition par le nombre total d'échantillons.

1. **Initialisation :** Choisir une variable aléatoire dans le réseau bayésien.
2. **Echantillonage :** Échantillonnez une valeur pour cette variable en utilisant sa distribution a priori. Cela représente une hypothèse sur la valeur de cette variable avant de prendre en compte des observations.
3. **Propagation :** Propagez la valeur échantillonnée dans le réseau bayésien en respectant les probabilités conditionnelles définies par la structure du graphe. Cela signifie que vous échantillonnez les valeurs des variables parentes en fonction des valeurs déjà échantillonées.
4. **Répétez** les étapes 2 et 3 jusqu'à ce que toutes les variables aléatoires du réseau bayésien aient une valeur échantillonnée. Ou bien jusqu'à ce que le nombre d'échantillons soit suffisant.

Un échantillon est donc une assignation de valeurs à toutes les variables aléatoires du réseau bayésien. On peut ensuite compter le nombre d'échantillons qui satisfont une condition et calculer la probabilité en divisant ce nombre par le nombre total d'échantillons.



Il est fort probable que cet algorithme ne soit pas précis, ou qu'il ne prenne pas en compte tous les cas de figure.

Remarque: C'est comme si on apprenait à partir des données.

- We'll get a bunch of samples from the BN:

$C, \neg S, \neg R, W$
 $C, S, \neg R, W$
 $\neg C, S, \neg R, \neg W$
 $C, \neg S, R, \neg W$
 $\neg C, \neg S, \neg R, W$

- If we want to know $P(W)$

- We have counts $\langle w:4, \neg w:1 \rangle$
- Normalize to get $P(W) = \langle w:0.8, \neg w:0.2 \rangle$
- This will get closer to the true distribution with more samples
- Can estimate anything else, too
 - $P(C | w)? P(C | w, r)? P(C | \neg w, \neg r)?$

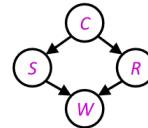


FIGURE 7.10 – Exemple avec Prior Sampling

7.3.2 Rejection Sampling

La différence avec l'échantillonage par prior est que l'on rejette les échantillons qui ne satisfont pas la condition de nos observations.

1. **Initialisation** : Choisir une variable aléatoire dans le réseau bayésien.
2. **Echantillonage** : Échantillonnez une valeur pour cette variable en utilisant sa distribution a priori. Cela représente une hypothèse sur la valeur de cette variable avant de prendre en compte des observations.
3. **Propagation** : Propagez la valeur échantillonnée dans le réseau bayésien en respectant les probabilités conditionnelles définies par la structure du graphe. Cela signifie que vous échantillonnez les valeurs des variables parentes en fonction des valeurs déjà échantillonées.
4. **Rejet** : Si l'échantillon ne satisfait pas la condition, alors on le rejette.
5. **Répétez** les étapes 2 et 3 jusqu'à ce que toutes les variables aléatoires du réseau bayésien aient une valeur échantillonnée. Ou bien jusqu'à ce que le nombre d'échantillons soit suffisant.

Exemple: Si nous cherchons à calculer $P(B|j, m)$, alors nous rejetons tous les échantillons qui ne satisfont pas j et m . Car on s'en fout de $\neg j$ et $\neg m$.

7.3.3 Likelihood Weighting

Le problème avec l'échantillonage par rejet est que l'on rejette beaucoup d'échantillons. Si les données que l'on veut sont rares, alors on va rejeter beaucoup d'échantillons et l'algorithme sera très lent.

L'idée est de générer des échantillons qui satisfont les observations. On **force** les échantillons à satisfaire les observations. Le problème est que les échantillons ne sont plus tirés de manière aléatoire donc la distribution des échantillons n'est plus la même, elle est **biaisée**.

Pour résoudre ce problème, on va **pondérer** les échantillons avec la probabilité d'observer la variable selon ses parents. Plus un échantillon est probable, plus il aura de poids et inversement.

- **Input** : Les observations $E_i = e_i$.
- weight $\leftarrow 1$
- **Pour** chaque variable aléatoire X_i dans le réseau bayésien :
 - Si X_i est une variable observée $E_i = e_i$:
 - $x_i \leftarrow e_i$
 - weight \leftarrow weight $\times P(X_i | Parents(X_i))$
 - Sinon :
 - $x_i \leftarrow$ échantillon de $P(X_i | Parents(X_i))$
- **Output** : x_1, \dots, x_n et weight

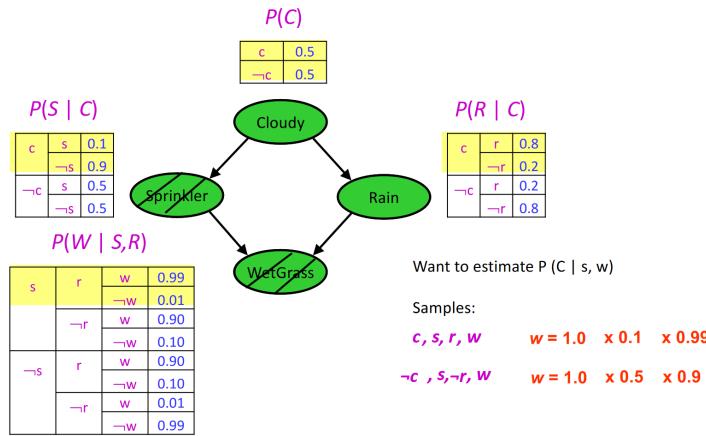


FIGURE 7.11 – Execution de Likelihood algorithme

7.3.4 Gibbs Sampling

Cette méthode résoud le problème du *Likelihood Weighting* qui est qu'il risque d'y avoir un biais lorsque les évidences sont les variables aléatoires qui ont le plus de parents.

L'idée est de fixer les valeurs des variables observées et de générer aléatoirement des valeurs pour les autres variables. De cette manière, on a un échantillon de départ. Pour toutes les variables qui sont pas observées, on va échantillonner une valeur en fonction des valeurs des variables déjà échantillonnées tout en gardant les valeurs des variables observées fixes. Chaque valeur échantillonnée est une hypothèse sur la valeur des autres variables aléatoires.

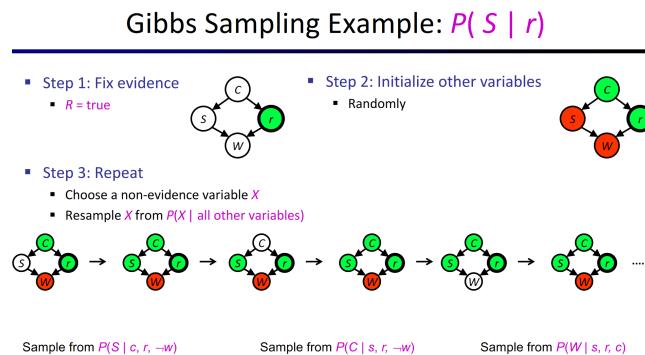


FIGURE 7.12 – Execution de l'algorithme de Gibbs

8 Modèle de Markov

Définition 8.1. Modèles de Markov

Les **Modèles de Markov** sont une sorte de réseaux bayésiens qui décrivent l'évolution d'un système dans le temps. Ils sont composés de variables aléatoires X_1, X_2, \dots, X_n qui représentent l'état du système à différents moments dans le temps. Les variables aléatoires sont liées par des probabilités de transition $P(X_t|X_{t-1})$ qui décrivent la probabilité de passer d'un état à l'autre, comment l'état du système évolue dans le temps.

Ces modèles sont utiles pour prédire l'état d'un système à un moment donné.

Théorème 8.1. Processus Markovien

Soit X_1, X_2, \dots, X_n une séquence de variables aléatoires. Si la propriété de Markov est respectée, alors

$$P(X_t|X_{t-1}, X_{t-2}, \dots, X_1) = P(X_t|X_{t-1})$$

Cela signifie que **le future** est **indépendant** du **passé**, étant donné le **présent**.

Théorème 8.2. Proccessus Stationnaire

Soit X_1, X_2, \dots, X_n une séquence de variables aléatoires. Si la propriété de Markov est respectée, alors

$$P(X_t|X_{t-1}) = P(X_{t+1}|X_t)$$

Cela signifie que la probabilité de transition est la même pour tous les instants. On dit que le modèle est **stationnaire**.

Remarque: Lorsque l'instant t est indépendant des instants $t-2, t-3, \dots, 1$, sachant $t-1$, on dit que le modèle est d'ordre 1. Il est possible d'avoir des modèles d'ordre supérieur, dans lequel l'instant t dépend des k instants précédents.

Pour calculer la probabilité de distribution conjointe d'un modèle de Markov, il faut utiliser la propriété de Markov et la règle de la chaîne.

$$\begin{aligned} P(X_1, X_2, \dots, X_n) &= P(X_1)P(X_2|X_1)P(X_3|X_2, X_1) \dots P(X_n|X_{n-1}, \dots, X_1) \\ &= P(X_1) \prod_{t=2}^n P(X_t|X_{t-1}) \end{aligned}$$

8.1 Le modèle qu'on ne trouve pas

La différence avec un modèle de Markov, c'est que un modèle de Markov caché est un modèle dans lequel après chaque état, une observation est générée. On peut alors utiliser les observations pour inférer l'état du système.

Nous supposons également que l'observation est générée **uniquement par** l'état courant du système.

$$P(E_t|X_{1:t}, E_{1:t-1}) = P(E_t|X_t)$$

Sachant que l'état du système est connu, l'observation est indépendante des observations précédentes ainsi que des états précédents.

Exemple: Un gardien de sécurité passe un mois dans un édifice sous-terrain, sans sortir. Chaque jour, son directeur arrive avec ou sans parapluie. Le gardien veut inférer la possibilité qu'il ait plu ou non en fonction de la séquence des observations du parapluie.

Voici le modèle de Markov pour ce problème.

- **Variables** : $X_t = R_t$ qui représente *Rain* et $E_t = U_t$ qui représente *Umbrella*.
- Dépendances entre les variables, càd Modèle de Markov :

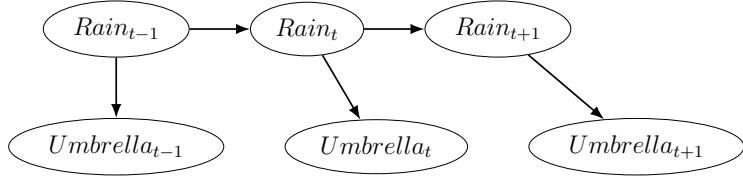


FIGURE 8.1 – Modèle de Markov

- Modèle de transitions : $P(R_t|R_{t-1})$ et $P(U_t|R_t)$ *Stationnaire*

Nous voyons bien que l'état du système à un moment donné ne dépend que de l'état précédent. Et que les observations sont générées uniquement par l'état courant du système.

Pour calculuer la probabilité de distribution conjointe d'un modèle de Markov caché, il faut utiliser la propriété de Markov et la règle de la chaîne.

$$\begin{aligned} P(X_1, X_2, \dots, X_n, E_n) &= P(X_1)P(X_2|X_1)P(X_3|X_2, X_1) \dots P(X_n|X_{n-1}, \dots, X_1)P(E_n|X_n) \\ &= P(X_1) \prod_{t=2}^n P(X_t|X_{t-1})P(E_t|X_t) \end{aligned}$$

8.2 Inférence

Il y a plusieurs problèmes d'inférence que l'on peut résoudre avec les modèles de Markov cachés.

- **Filtering** : Calculer la distribution de probabilité de l'état courant du système, sachant toutes les observations jusqu'à maintenant.

$$P(X_t|E_{1:t})$$

- **Prediction** : Calculer la distribution de probabilité de l'état futur du système, sachant toutes les observations jusqu'à maintenant. Sans tenir compte des observations futures.

$$P(X_{t+k}|E_{1:t})$$

- **Smoothing** : Calculer la distribution de probabilité de l'état passé du système, sachant toutes les observations jusqu'à maintenant. Meilleure estimation de l'état passé. *Learning*

$$P(X_{t-k}|E_{1:t})$$

- **Most Likely Explanation** : Calculer la séquence d'états **la plus probable**, sachant toutes les observations jusqu'à maintenant.

$$\arg \max_{X_{1:t}} P(X_{1:t}|E_{1:t}) = \arg \max_{X_{1:t}} P(X_{1:t}, E_{1:t}) / P(E_{1:t}) = \arg \max_{X_{1:t}} P(X_{1:t}, E_{1:t})$$

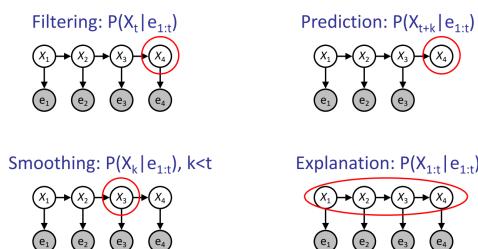


FIGURE 8.2 – Inférence dans un modèle de Markov

8.3 Chaines de Markov

Les chaines de Markov sont un cas particulier des modèles de Markov. Elles sont composées d'un ensemble d'états $S = s_1, s_2, \dots, s_n$ et d'une matrice de transition T qui décrit la probabilité de passer d'un état à l'autre.

$$T_{ij} = P(X_t = s_j | X_{t-1} = s_i)$$

(même définition que pour les modèles de Markov, c'est juste moins général)

9 Réseaux de décisions

Définition 9.1. Réseaux de décisions

Un réseau de décision, également connu sous le nom de réseau de décision bayésien (RDB), est un modèle graphique probabiliste qui étend le concept de réseau bayésien pour inclure la prise de décision dans un environnement incertain. Un réseau de décision intègre des éléments tels que des actions et des utilités pour modéliser des situations où des décisions doivent être prises dans un contexte de probabilités et d'incertitude. Il est composé de trois types de noeuds : les noeuds de décision, les noeuds de chance et les noeuds d'utilité.

- **Nœuds de décision** : Les noeuds de décision sont des noeuds carrés qui représentent les actions à prendre.
- **Nœuds de chance** : Les noeuds de chance sont des noeuds ovales qui représentent les événements incertains. Comme dans les réseaux bayésiens.
- **Nœuds d'utilité** : Les noeuds d'utilité sont des noeuds hexagonaux qui représentent les fonctions d'utilité. Il est souvent fils d'un noeud de décision et/ou d'un noeud de chance.

Son but est de trouver la meilleure action.

Voici l'algorithme pour trouver la meilleure action.

- Instancier les noeuds de chance avec les valeurs observées (n'importe laquelle). Instancier les observations.
- Pour chaque action
 - Instancier action par une valeur. (celle qui nous intéresse).
 - Pour chaque noeuds d'utilité, calculer la probabilité à postériori de ses parents selon les observations.
 - Pour chaque noeuds d'utilité, calculer la valeur de l'utilité attendue.
- Choisir l'action qui maximise l'utilité attendue.

| Exemple:

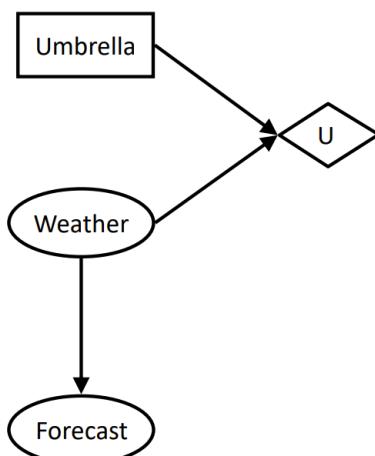


FIGURE 9.1 – Réseau de décision

Umbrella est l'action de prendre, ou non, un parapluie. Il y a deux nœuds de chance : le premier Weather est le nœud de chance de la météo, qui peut être soit pluvieux, soit ensoleillé. Le second, Forecast, est le nœud de chance de la prévision météorologique, qui peut être soit pluvieux, soit ensoleillé. Il y a un nœud d'utilité, U , qui représente le coût de prendre un parapluie.

Pour calculer la probabilité à postériori de Weather, $P(W|e, a)$.

Pour calculer l'utilité attendue, $EU(a) = \sum_w P(w|e, a)U(w, a)$. Car en effet, U dépend de Weather et de Umbrella.

Nous pouvons représenter un réseau de décision par un arbre de "résultat" (outcome tree). Ces arbres sont utilisés pour représenter l'algorithme **Expectimax**. Sauf que cette fois pour avoir les probabilités des noeuds de chance, il faut calculer la probabilité à postériori dans un réseau bayésien.

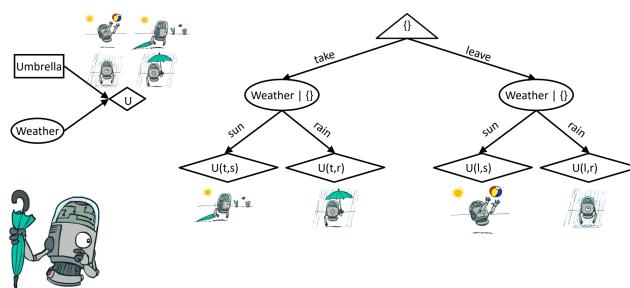


FIGURE 9.2 – Arbre de résultat

9.1 Value of Perfect Information

Définition 9.2. Value of Perfect Information

La valeur de l'information parfaite est la différence entre l'utilité attendue maximale avec l'information parfaite et l'utilité attendue maximale sans l'information parfaite.

$$VPI = EU_{max}^{PI} - EU_{max}^{NPI} \quad (12)$$

où PI est l'information parfaite et NPI est sans l'information parfaite.

Remarque: De ce que j'ai compris, l'information parfaite correspond à la révélation d'un noeud de chance dans le réseau de décision.

Imaginons que nous avons les observations $E = e$. Pour calculer l'utilité on a :

$$MEU(e) = \max_a \sum_w P(s|e)U(s, a)$$

où s est l'état du monde.

Si on apprend qu'il existe une information $E' = e'$ qui est rend plus précis notre calcul de l'utilité, on a :

$$MEU(e, e') = \max_a \sum_w P(s|e, e')U(s, a)$$

Cependant, on n'a pas cette information. Celle-ci est aussi incertaine. Il faut donc calculer la distribution de probabilité de E' .

$$MEU(e, E') = \sum_{e'} P(e'|e)MEU(e, e')$$

Le **VPI** est donc l'information de combien l'utilité attendue augmente si on a l'information E' qui nous est révélée.

$$VPI = MEU(e, E') - MEU(e)$$

Voici les propriétés du **VPI** :

- $VPI \geq 0$
- VPI est non-additif. $VPI(E_1, E_2|e) \neq VPI(E_1|e) + VPI(E_2|e)$
- VPI Ne dépend pas de l'ordre dans lequel les informations sont révélées. $VPI(E_1, E_2|e) = VPI(E_2, E_1|e)$

Si il y a une observation Z qui est conditionnellement indépendante à $Parent(U)$ sachant les autres observations ALORS le **VPI** est égal à 0.

10 Markovian Decision Processes

Définition 10.1. Markovian Decision Processes

Un **Markovian Decision Process** (MDP) est un modèle de processus de décision séquentiel dans lequel les états sont Markoviens. Un MDP est défini par un tuple $\langle S, A, T, R, \gamma \rangle$ où :

- S est un ensemble fini d'états ;
- A est un ensemble fini d'actions ;
- $T : S \times A \times S \rightarrow [0, 1]$ est la fonction de transition. On a que $T(s, a, s') = P(s'|s, a)$ qui est la probabilité de passer de l'état s à l'état s' en effectuant l'action a ;
- $R : S \times A \times S \rightarrow \mathbb{R}$ est la fonction de récompense ;
- S_0 est l'état initial ;
- S_T est l'état terminal ;

Nous sommes donc dans un environnement **Non-Déterministe**, **Séquentiel**, **Statique**, **Discret** et **Complètement observable**. Une action a n'a pas toujours le même effet dans un état s donné. En effet, il y a une probabilité $T(s, a, s')$ que l'action a dans l'état s nous amène à l'état s' .

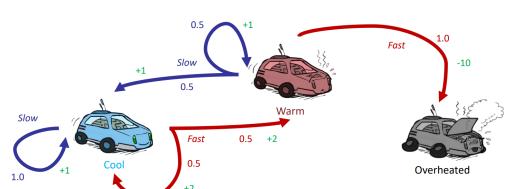
Dans une recherche déterministe, nous pouvons toujours déterminer l'état suivant. Par ce fait nous voulions établir un **plan** qui est une séquence d'actions qui nous amène à l'état final. Dans un MDP, nous ne pouvons pas établir un plan car nous ne pouvons pas déterminer l'état suivant à 100% de certitude. Nous devons donc établir une **politique** qui est une fonction qui nous dit quelle action prendre dans un état donné.

Définition 10.2. Politique

Une **politique** est une fonction $\pi : S \rightarrow A$ qui nous dit quelle action prendre dans un état donné. La politique **optimale** est la politique qui maximise l'utilité, notée π^* .

Remarque: Un plan n'est pas une bonne idée dans un environnement non-déterministe car il peut y avoir des changements dans l'environnement qui rendent le plan **obsolète**.

Exemple: Imaginons une voiture qui peut se déplacer soit rapidement, soit lentement. En fonction de ses actions, elle est soit *Cool*, *Warm* ou *Overheated*. Voici la représentation via un automate :



s	a	s'	T(s,a,s')	R(s,a,s')
	Slow		1.0	+1
	Fast		0.5	+2
	Fast		0.5	+2
	Slow		0.5	+1
	Slow		0.5	+1
	Fast		1.0	-10
	(end)		1.0	0

FIGURE 10.1 – Voiture de course (kachow)

Nous pouvons alors représenter l'arbre de recherche de la voiture de course comme suit :

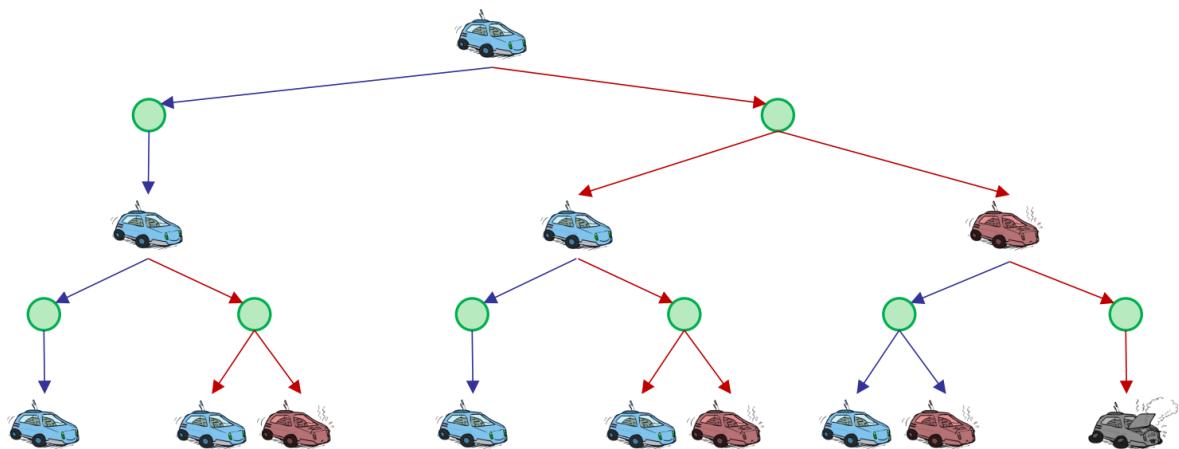


FIGURE 10.2 – Arbre de recherche

Nous pouvons faire un lien entre la représentation de l'arbre et l'arbre de l'algorithme **Expectimax** vu précédemment. En effet, nous pouvons voir que les **nœuds chance** sont les nœuds où la voiture peut changer d'état en fonction d'une action.

10.1 Hyper-Paramètres

Définition 10.3. Discount Factor

Le **discount factor** γ est un hyper-paramètre qui représente l'importance des récompenses futures. Plus γ est proche de 1 ou supérieur, plus les récompenses futures sont importantes. Plus γ est proche de 0, plus les récompenses futures sont négligeables.

- $\gamma = 0$: **Myopic** (court-termiste) ;
- $\gamma = 1+$: **Far-sighted** (long-termiste) ;

Si $\gamma = 1$, les récompenses futures sont aussi importantes que les récompenses actuelles.

A chaque étape de la recherche, nous multiplions la récompense par γ^k où k est le nombre d'étapes

Définition 10.4. Living Reward

Le **living reward** est une récompense négative qui est donnée à chaque étape de la recherche. Cela permet d'éviter que l'agent ne reste dans un état pendant trop longtemps.

Définition 10.5. Noise

Le **noise** est un hyper-paramètre qui représente la probabilité que l'action choisie ne soit pas celle effectuée. Cela permet de représenter le fait que l'agent ne peut pas toujours effectuer l'action qu'il veut. Par exemple, si l'agent veut aller à gauche, il peut y avoir un obstacle qui l'en empêche et va donc aller à droite. Le noise permet de représenter ce genre de situation.

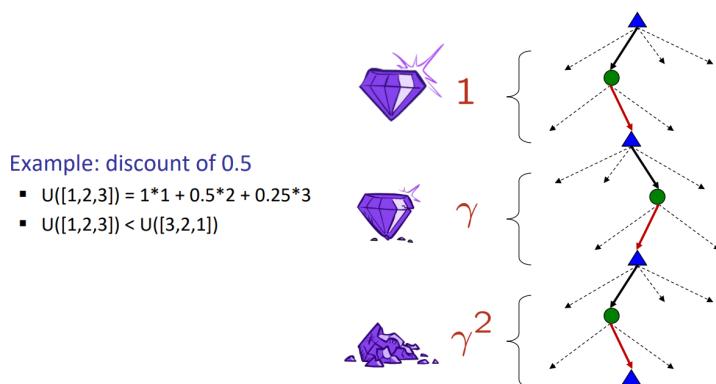


FIGURE 10.3 – Illustration du discount factor. $U()$ = l'utilité = $Reward \times \gamma^k$

L'utilité d'une politique π est définie comme suit :

$$U^\pi(s) = \mathbb{E} \left[\sum_{k=0}^{\infty} \gamma^k R(s_k, a_k, s_{k+1}) \right] \quad (13)$$

C'est-à-dire que l'utilité d'une politique est la somme des récompenses futures pondérées par le discount factor.

10.2 Equation de Bellman

10.2.1 Quantité optimale

Définition 10.6. Utilité d'un état

Nous définissons l'utilité d'un état s comme la valeur $V^*(s)$. C'est l'utilité espérée à partir de s en suivant la politique optimale π^* .

$$V^*(s) = \max_a Q^*(s, a)$$

Définition 10.7. Utilité d'un q-état

Nous définissons l'utilité d'un q-état (s, a) comme la valeur $Q^*(s, a)$. C'est l'utilité espérée à partir de s en suivant la politique optimale π^* et en effectuant l'action a .

$$Q^*(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

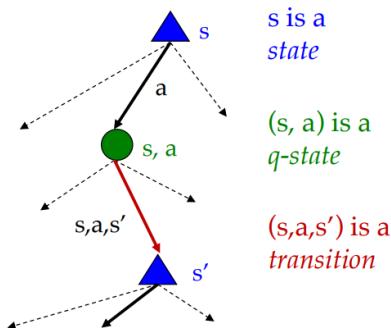


FIGURE 10.4

10.2.2 Value Iteration

Algorithm 5 Value Iteration

```

1: procedure VALUEITERATION( $S, A, T, R, \gamma, \epsilon$ )
2:    $V(s) \leftarrow 0$  for all  $s \in S$ 
3:   repeat
4:      $V_k \leftarrow V$  # Copie de  $V$ 
5:     for all  $s \in S$  do
6:        $V_k(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V(s')]$ 
7:     end for
8:      $V \leftarrow V_k$  # Mise à jour de  $V$ 
9:     until  $\max_{s \in S} |V(s) - V'(s)| < \epsilon$ 
10:    return  $V$ 
11: end procedure

```

L'algorithme est répété jusqu'à ce que la différence entre V et V' soit inférieure à ϵ . Soit ce que l'on appelle la **convergence**. L'algorithme va converger au bout d'un certain nombre d'itérations. En effet, plus le nombre d'itérations augmente, plus la différence entre V et V' diminue de part le discount factor γ qui devient de plus en plus petit.

Complexité : $O(|S|^2 \times |A|)$

10.2.3 Policy Extraction

Algorithm 6 Policy Extraction

```

1: procedure POLICYEXTRACTION( $S, A, T, R, \gamma, V$ )
2:    $\pi(s) \leftarrow \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V(s')]$ 
3:   return  $\pi$ 
4: end procedure

```

Cet algorithme permet d'extraire la politique optimale π^* à partir de V^* . En effet, nous pouvons voir que la politique optimale est la politique qui maximise l'utilité. Nous pouvons donc extraire la politique optimale à partir de V^* en choisissant l'action qui maximise l'utilité pour chaque état.

10.2.4 Policy Evaluation

Cet algorithme permet de calculer l'utilité d'une politique π donnée. Il va calculer l'utilité de chaque état en fonction de la politique π et de la fonction de transition T . Il y a moins de calculs à faire que pour l'algorithme de **Value Iteration** car nous n'avons pas besoin de calculer toutes les valeurs de V à chaque itération. Nous avons juste besoin l'utilité d'un état en fonction **d'une seule** action.

Algorithm 7 Policy Evaluation

```

1: procedure POLICYEVALUATION( $S, A, T, R, \gamma, \pi$ )
2:    $V(s) \leftarrow 0$  for all  $s \in S$ 
3:   repeat
4:      $V_k \leftarrow V$  # Copie de  $V$ 
5:     for all  $s \in S$  do
6:        $V_k(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V(s')]$ 
7:     end for
8:      $V \leftarrow V_k$  # Mise à jour de  $V$ 
9:     until  $\max_{s \in S} |V(s) - V'(s)| < \epsilon$ 
10:    return  $V$ 
11: end procedure

```

Nous pouvons voir que l'algorithme est similaire à l'algorithme de **Value Iteration** mais que nous n'avons pas besoin de calculer la somme des utilités de toutes les actions possibles. Nous avons juste besoin de calculer l'utilité d'une seule action. Cela permet de réduire le nombre de calculs à faire.

Complexité : $O(|S|^2)$

10.2.5 Policy Iteration

L'algorithme de **Policy Iteration** est un algorithme qui permet de trouver la politique optimale π^* à partir d'une politique initiale π_0 . Il va alterner entre les étapes de **Policy Evaluation** et **Policy Extraction** jusqu'à ce que la politique ne change plus. La différence avec l'algorithme de **Value Iteration** est que on commence avec une politique initiale π_0 et on va la modifier jusqu'à ce qu'elle soit optimale. On a donc moins de calculs à faire car on ne va pas devoir calculer toutes les valeurs de V à chaque itération.

Algorithm 8 Policy Iteration

```

1: procedure POLICYITERATION( $S, A, T, R, \gamma, \epsilon$ )
2:    $\pi(s) \leftarrow$  random policy
3:   repeat
4:      $V_{\pi_{k+1}} \leftarrow$  POLICYEVALUATION( $S, A, T, R, \gamma, \pi$ )
5:      $\pi_{i+1} \leftarrow$  POLICYEXTRACTION( $S, A, T, R, \gamma, V$ )
6:   until  $\pi$  does not change
7:   return  $\pi$ 
8: end procedure

```

Complexité : $O(|S|^2 \times |A|) + O(|S|^3)$

Cette algorithme est plus lent mais converge plus rapidement que l'algorithme de **Value Iteration** (*moins d'itérations*).

| **Remarque:** Voir exemple slide 22 cours 15.

11 Reinforcement Learning

Considérons toujours un MDP avec :

- des états $s \in S$
- des actions $a \in A$
- un modèle $T(s, a, s')$
- une fonction de récompense $R(s, a, s')$

Nous sommes toujours à la recherche d'une politique $\pi(s)$. Cependant, nous ne connaissons plus pour autant le modèle ni la fonction de récompense. Le but est d'apprendre quelles actions prendre dans quel état pour maximiser la récompense. Un schéma qu'il faut avoir en tête est le suivant :

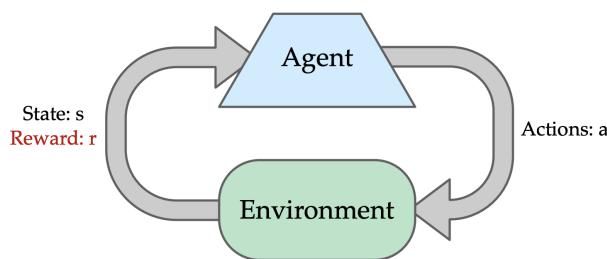


FIGURE 11.1 – Schéma de l'apprentissage par renforcement

L'idée est la suivante :

- On reçoit un feedback de l'environnement (reward).
- L'utilité de l'agent est définie par la fonction de récompense.
- L'agent doit apprendre à maximiser les récompenses attendues.
- Tous l'apprentissage est basé sur des observations d'output de l'environnement.

11.1 Passive Reinforcement Learning

Nous parlons de passive RL car nous avons une politique fixe $\pi(s)$, nous ne connaissons pas les transitions $T(s, a, s')$ ni les récompenses $R(s, a, s')$. Dans le cas du passive RL, on ne choisit pas les actions prises, on exécute simplement la politique et nous apprenons via l'expérience.



Ce n'est pas OFFLINE car l'apprenant prend des actions dans le monde.

11.1.1 Model-based Passive RL

Définition 11.1. Model-based Passive RL

Le but est d'apprendre le modèle MDP depuis les expériences et ensuite de résoudre le MDP appris.

Le model-based RL est donc divisé en deux étapes :

1. Apprendre le modèle MDP empirique
 - (a) Compter les résultats s' pour chaque transition (s, a) .
 - (b) Normaliser afin de donner une estimation $\hat{T}(s, a, s')$.
 - (c) Découvrir chaque $\hat{R}(s, a, s')$ quand on fait (s, a, s') .
2. Résoudre le MDP appris, en utilisant par exemple la value iteration.

Exemple: Imaginons que notre but est de prédire l'âge des étudiants en informatique. Nous avons vu précédemment comment trouver ceci grâce à l'équation suivante :

$$E[A] = \sum_a P(a) \cdot a$$

Maintenant, imaginons qu'on ait pas $P(A)$, on pourrait alors récolter un tas d'échantillons $[a_1, a_2, \dots, a_n]$ et voici la différence entre l'approche *model based* et *model free* :

$$\text{model-based} \rightarrow E[A] \approx \sum_a \hat{P}(a) \cdot a | \hat{P}(a) = \frac{\text{count}(a)}{n}$$

Le $\hat{P}(a)$ du *model-based* est éventuellement le bon modèle et c'est pour cela que ça fonctionne.

11.1.2 Model-free Passive RL

Définition 11.2. Model-free Passive RL

Renoncer à l'apprentissage du modèle MDP et apprendre directement la fonction de valeur V ou Q .

- V fait référence à value learning où on apprend la valeur d'une politique fixe. Elle est constituée de deux approches : l'**évaluation directe** (cf. 11.1.2.1) et le **TD learning** (cf. 11.1.2.2).
- Q fait référence à Q-learning où on apprend les Q-valeurs de la politique optimale. L'approche utilisée est une Q-version du TD learning.

11.1.2.1 Direct evaluation

Le **but** est de calculer les valeurs pour chaque état sous la politique π . L'**idée** est de faire la moyenne de l'ensemble des valeurs observées de l'échantillon, pour ce faire :

- On agit par rapport à la politique π .
- A chaque fois qu'on visite un état, on met à jour la somme des récompenses.
- On fait la moyenne sur beaucoup d'échantillons.

Exemple: Voici un exemple de direct evaluation :

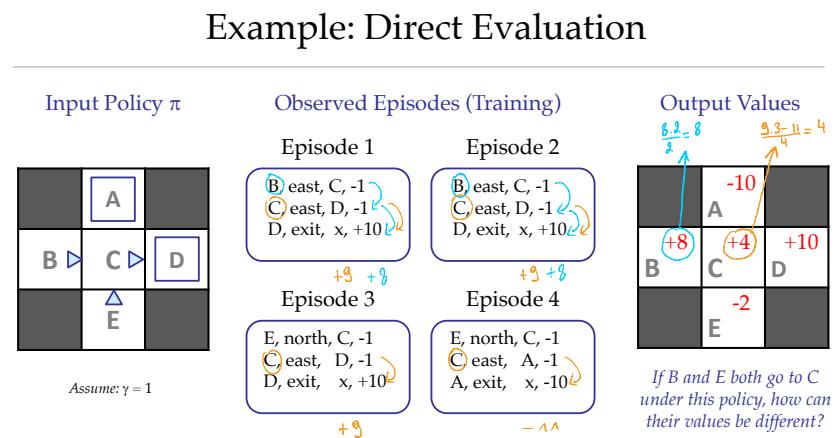


FIGURE 11.2 – Exemple de direct evaluation

Les avantages et les défauts de cette méthode sont les suivants :

- + Simple
- + On a pas besoin de connaitre ni T ni R.
- + Il finit par calculer les valeurs moyennes correctes en utilisant uniquement les transitions de l'échantillon.
- Il y a une perte d'information entre les connexions entre les états.
- Chaque état doit être appris indépendamment.
- Il faut beaucoup de temps pour apprendre.

11.1.2.2 Temporal Difference Value Learning

L'idée est d'apprendre de chaque expérience au fur et à mesure, il faut donc mettre à jour $V(s)$ à chaque fois qu'on expérimente une transition (s, a, s', r) . La politique est toujours fixe mais les valeurs des états vont devenir de plus en plus précises. Donc :

- Sample of $V(s)$: sample = $R(s, \pi(s), s') + \gamma V^\pi(s')$
- Update to $V(s)$: $V(s) \leftarrow (1 - \alpha)V^\pi(s) + \alpha$ sample
- En transformant l'équation : $V(s) \leftarrow V^\pi(s) + \alpha(\text{sample} - V^\pi(s))$

La dernière équation se lit de la façon suivante, on prend la valeur actuelle de l'état ($V^\pi(s)$) et on ajoute à celle-ci une petite fraction (α) de la différence entre la récompense observée (sample) et la valeur actuelle de l'état ($V^\pi(s)$). De plus, le fait de faire une mise à jour dynamique, nous permet d'aller consulter la valeur de chaque état à tout moment et de la mettre à jour directement.

Le fait de faire une mise à jour de l'interpolation en cours ($\bar{x}_n = (1 - \alpha) \cdot \bar{x}_{n-1} + \alpha \cdot x_n$) permet de rendre les états récents plus importants tandis que les états plus anciens sont moins importants. Si le taux d'apprentissage α diminue, les moyennes convergent.

Exemple: Voici un exemple de TD learning :

Example: Temporal Difference Value Learning

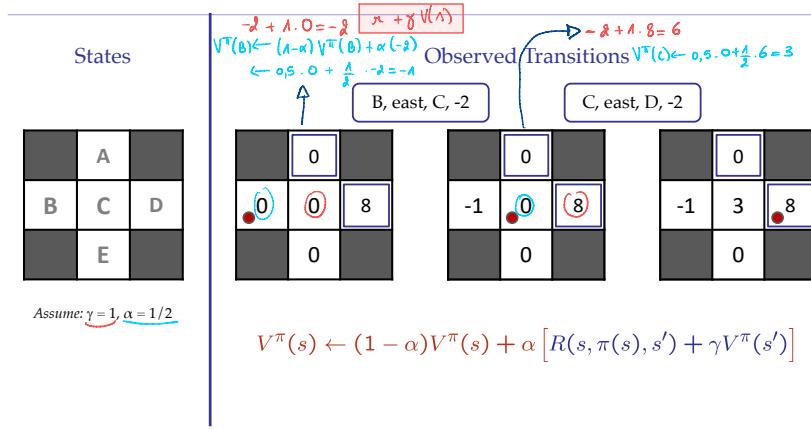


FIGURE 11.3 – Exemple de TD learning

Le **problème** avec le TD learning est que c'est un moyen sans modèle d'évaluer les politiques en imitant les mises à jour de Bellman avec des moyennes d'échantillons en cours d'exécution. Si on veut transformer les valeurs en une nouvelle politique, on est perdus, effectivement, on a besoin du modèle pour trouver une nouvelle politique :

$$\begin{aligned}\pi(s) &= \max_a Q(s, a) \\ Q(s, a) &= \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V(s')]\end{aligned}$$

11.1.3 Sample-based Policy Evaluation

Dans le cas où nous sommes dans un état s et qu'en suivant la politique $\pi(s)$, nous arrivons dans un état s' , et que nous appliquons à nouveau la politique $\pi(s)$, et depuis cet état on choisit aléatoirement à quel état s' on arrive. Les équations de Bellman nous permettent de trouver les valeurs des états de façon efficace :

$$\begin{aligned}V_0^{\pi}(s) &= 0 \\ V_{k+1}^{\pi}(s) &= \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^{\pi}(s')]\end{aligned}\tag{14}$$

Dans ce cas, nous exploitons clairement le lien entre les états, mais nous devons connaître T et R .

Pour faire sans, nous pouvons prendre un ensemble d'échantillons (en faisant les actions) et en faisant la moyenne.

$$\begin{aligned}\text{sample}_1 &= R(s, \pi(s), s'_1) + \gamma V_k^{\pi}(s'_1) \\ \text{sample}_2 &= R(s, \pi(s), s'_2) + \gamma V_k^{\pi}(s'_2) \\ &\vdots \\ \text{sample}_n &= R(s, \pi(s), s'_n) + \gamma V_k^{\pi}(s'_n)\end{aligned}$$

De cette manière nous évaluons l'équation (14) en utilisant des échantillons. En équation cela donne :

$$V_{k+1}^{\pi}(s) \leftarrow \frac{1}{n} \sum_{i=1}^n \text{sample}_i$$

11.2 Active Reinforcement Learning

La différence avec le passive RL est que nous pouvons choisir les actions à prendre. L'apprenant prend désormais des décisions.

11.2.1 Q-learning

Value iteration trouve des valeurs successives (limitées par une profondeur max) mais les **Q-values** sont plus pratiques, nous pouvons réécrire les équations de value iteration ($V_{k+1}(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$) en termes de Q-values :

- On commence avec $Q_0(s, a) = 0$.
- Pour un Q_k donné, on calcule Q_{k+1} pour tous les q-états :

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \max_{a'} Q_k(s', a')]$$

La différence entre les deux est que dans le cas des q-values, on commence par une moyenne tandis que dans le cas de value iteration, on commence par un max. L'utilisation d'échantillons n'est donc pas possible dans value iteration. Sauf qu'avec les q-values, c'est possible, l'équation écrite au dessus représente en fait la **Q-value iteration**.

A partir d'un échantillon (s, a, s', r) , on peut mettre à jour $Q(s, a)$:

$$\text{sample} = R(s, a, s') + \gamma \max_{a'} Q(s', a') \quad (15)$$

On voit qu'on a plus d'évaluation de politique. On peut intégrer la nouvelle estimation dans la moyenne dynamique :

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \text{ sample}$$

Voici les propriétés de Q-learning :

- Q-learning converge vers les Q-valeurs optimales même si on agit de manière suboptimale.
- C'est une méthode d'apprentissage "off-policy"

Cependant, il faut faire attention à :

- Parcourir suffisamment longtemps.
- Il faut parfois donner une valeur très petite à α mais il ne faut pas le diminuer trop rapidement.
- Il n'y a pas vraiment d'importance à la façon dont on choisit les actions.

Remarque: Voici ce qui a été vu jusque maintenant :

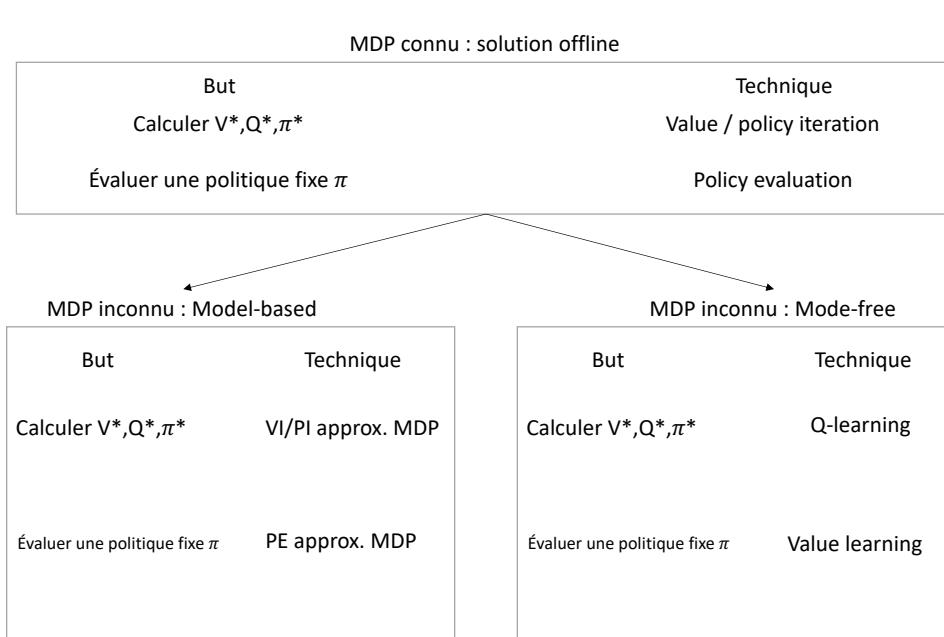


FIGURE 11.4 – Résumé de ce qui a été vu jusque maintenant

11.3 Exploration vs Exploitation

Exemple: Imaginons que vous alliez toujours dans le même restaurant qui est très bien et que vous adorez y aller. Mais un jour, un tout nouveau restaurant ouvre juste à côté, est-ce que vous allez continuer à aller dans le même restaurant ou est-ce que vous allez essayer le nouveau restaurant ? Aller tester le nouveau restaurant est risqué car il pourrait être moins bon que l'ancien. C'est le dilemme entre l'exploration et l'exploitation.

11.3.1 ϵ -greedy

C'est le plus simple, on choisit entre prendre une action aléatoire (avec une probabilité de ϵ) ou alors suivre la politique (avec une probabilité, plus élevée, de $1 - \epsilon$).

Dans ce cas, si l'algorithme tourne suffisamment longtemps, nous sommes sûrs d'explorer tous les états et les q-values vont converger vers les valeurs optimales. Afin d'avoir ces valeurs optimales, il faut que le ϵ diminue dans le temps afin de converger vers une politique optimale.

11.3.2 Fonctions d'exploration

L'idée ici se base (plutôt que d'explorer de façon aléatoire) sur le fait d'explorer des états qui sont pas encore bien connus. Mais si on connaît bien un état, cela ne sert à rien de revisiter cet état.

Pour ce faire, on utilise une fonction d'exploration de la forme suivante : $f(u, n) = u + \frac{k}{n}$ où u est la valeur de l'état et n est le nombre de fois qu'on a visité l'état et k est une constante. Nous remarquons donc qu'au plus un état a été visité, la fonction diminue puisque le terme en $\frac{k}{n}$ diminue ($n >>$).

La q-value de l'équation (15) s'écrit désormais :

$$Q(s, a) \leftarrow_{\alpha} R(s, a, s') + \gamma \max_{a'} f(Q(s', a'), N(s', a'))$$

Où \leftarrow_{α} dit qu'on fait une mise à jour avec un poids, on a déjà une valeur Q , on en a une nouvelle et donc on va combiner l'**ancienne** $+(\alpha \times)$ la **nouvelle**. Dès lors, on prend en compte les états qui n'ont pas été visité beaucoup en leur donnant une valeur plus élevée, si on a des actions

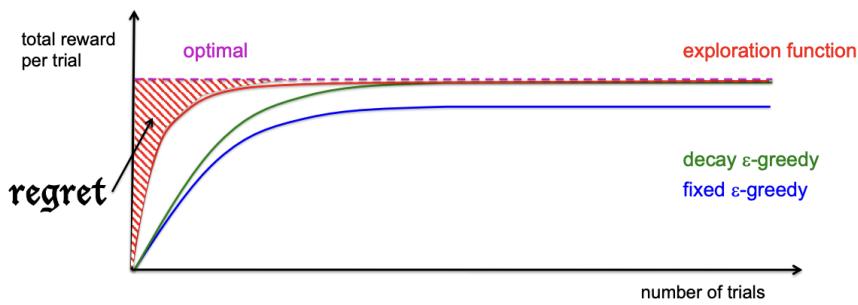


FIGURE 11.5 – Exemple de regret

dans s' qu'on a pas encore beaucoup prises, si c'est le cas, alors la valeur sera plus grande.

11.3.3 Regret

Si on a appris deux politiques optimales selon deux algorithmes différents, comment savoir laquelle est la meilleure ?

Il faut prendre en compte non seulement la récompense finale mais on va regarder toutes les récompenses obtenus au fur et à mesure de l'algorithme. On regardera par exemple quel algorithme a trouvé la politique optimale de façon la plus rapide. La notion de regret intervient dans cette comparaison, et donc une exploration aléatoire aura un regret bien plus élevé qu'une exploration basée sur des fonctions d'exploration. Le regret ne sera jamais égal à 0.

11.4 Approximate Q-learning

Dans le cas de q-learning, on garde une table avec toutes les q-values. Cependant, dans des situations réalistes, on ne peut pas faire ceci, car trop d'états par exemple. Le but est d'apprendre d'un nombre d'état minimum et d'ensuite faire une généralisation.

Exemple: Si on a déjà été dans un building par une certaine porte, on peut essayer d'entrer dans un autre building même si on ne connaît pas et qu'on a jamais vu la porte de ce nouveau building.

La solution c'est de décrire un état par un vecteur de features (propriétés). Pour combiner les valeurs de ces features, on va utiliser des fonctions de valeur linéaires. On pourrait faire ceci sur le value learning ou sur le q-learning mais ici, on va se concentrer sur le q-learning :

$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \dots + w_n f_n(s, a) \quad (16)$$

Donc ici, on a un ensemble de features (f_1, f_2, \dots, f_n) et un ensemble de poids (w_1, w_2, \dots, w_n) . On attend de q-learning d'apprendre ces poids, de façon qu'une fois que les poids sont appris, les q-values soient plus ou moins optimales.



Il se peut que deux états aient les mêmes features mais des q-values différentes.

Pour une transition (s, a, r, s') , on a la différence qui vaut $[r + \gamma \max_{a'} Q(s', a')] - Q(s, a)$. Dès lors :

$$\begin{aligned} Q(s, a) &\leftarrow Q(s, a) + \alpha[\text{différence}] \\ w_i &\leftarrow w_i + \alpha[\text{différence}] f_i(s, a) \end{aligned}$$

Exemple: Voici un exemple sur le jeu Pac-Man :

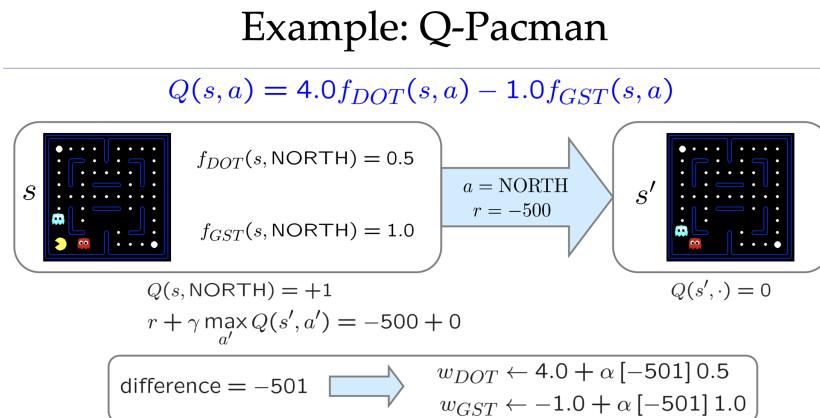


FIGURE 11.6 – Exemple d’approximate q-learning

Avec f_{DOT} qui est le feature du joueur par rapport à un point et f_{GST} qui est le feature du joueur par rapport à un fantôme.

12 Machine Learning

La but du machine learning est d’acquérir un modèle à partir de données ou d’expériences. Par exemple, en apprenant des paramètres (probabilités), en apprenant des structures (réseaux bayésiens)…

12.1 Classification

Étant donné un point d’entrée x de données avec certains features, on veut prédire une sortie y qui est une classe. Voici comment résumer cela en une ligne :

$$x(\text{input}) \rightarrow \text{features(attributs de } x) \rightarrow y(\text{output})$$

Le machine learning intervient entre l’étape de features et l’étape de prédiction. L’idée d’un algorithme de machine learning est d’apprendre les patterns entre les features et les classes depuis des données. Les données (d’entraînement) sont des exemples qui ont été préalablement classifiés.

Exemple: Spam Filter :

- input : un mail
- output : spam ou non spam

Le but est d’avoir en amont un grand nombre de mails labélisés en tant que spam ou non spam (cette action doit être faite par quelqu’un). Ensuite, on rajoute des features :

- On peut rajouter des mots clés (free, money, ...)
- Des patterns de texte (trop de majuscules, trop de points d’exclamation, ...)
- ...

12.2 Model-based classification

Un peu comme dans la section précédente, il y a une distinction entre les *model – free* et les *model – based*. Donc ici, le but est de construire un modèle où les classes d’output et les features d’input sont des variables aléatoires. Il y aura des connexions entre les variables.

12.2.1 Naïve Bayes Model

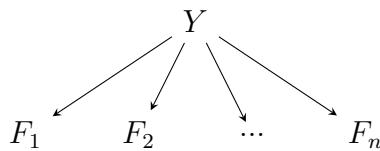


FIGURE 12.1 – Réseau de Bayes naïf

Y est la classe, F_1, F_2, \dots, F_n sont les features. Les tables de probabilités sont les suivantes :

- $P(Y)$: probabilité que chaque classe apparaisse sans savoir les features.
- $P(F_i|Y)$: Une table par feature, probabilité de distribution d'un feature étant donné une classe.

1. Pour l'entraînement :

On utilise un jeu de données pour apprendre les tables de probabilités, pour estimer $P(Y)$, on regarde le nombre de fois qu'il apparaît dans le jeu de données. Pour estimer $P(F_i|Y)$, on regarde comment la classe affecte le feature.

2. Pour la classification :

On instantie toutes les features, on connaît les input features. Pour $P(Y|F_1, F_2, \dots, F_n)$, on utilise un algorithme d'inférence pour calculer.

Pour un réseau de Bayes naïf, on a :

$$P(Y, F_1, \dots, F_n) = P(Y) \prod_{i=1} P(F_i|Y)$$

- $P(Y)$: contient $|Y|$ paramètres.
- $P(Y, F_1, \dots, F_n)$: contient $|Y| \times |F|^n$ paramètres.
- $P(F_i|Y)$: contient $n \times |Y| \times |F|$ paramètres.

Exemple: Réseau de Bayes naïf pour la reconnaissance de chiffre :

On suppose que toutes les features sont indépendantes de la classe. Ce n'est pas logique mais cela fonctionne très bien. Dans le cas de la reconnaissance des chiffres, il y a un feature F_{ij} pour chaque position dans la grille i, j . Par exemple pour 1, on a :

$$1 \rightarrow \langle F_{0,0} = 0, F_{0,1} = 0, F_{0,2} = 1, F_{0,3} = 1, F_{0,4} = 0 \dots F_{15,15} = 0 \rangle$$

Le modèle de Bayes naïf sera donc :

$$P(Y|F_{0,0}, F_{0,1}, \dots, F_{15,15}) \propto P(Y) \prod_{i,j} P(F_{i,j}|Y)$$

Naïve Bayes for Digits: Conditional Probabilities

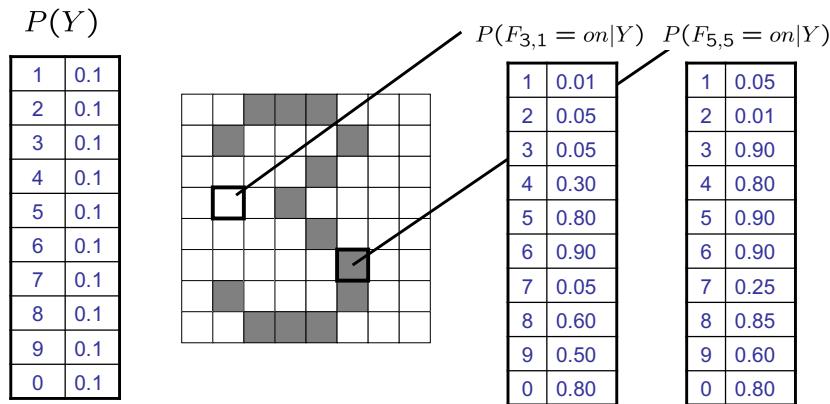


FIGURE 12.2 – Probabilités conditionnelles pour la reconnaissance de chiffre

12.2.2 Inference for Naïve Bayes

Le but est de calculer $P(Y|F_1, F_2, \dots, F_n)$ mais on sait qu'on peut utiliser la probabilité jointe :

$$P(Y, f_1 \dots f_n) = \begin{bmatrix} P(y_1, f_1 \dots f_n) \\ P(y_2, f_1 \dots f_n) \\ \vdots \\ P(y_n, f_1 \dots f_n) \end{bmatrix} \Rightarrow \begin{bmatrix} P(y_1) \prod_i P(f_i|y_1) \\ P(y_2) \prod_i P(f_i|y_2) \\ \vdots \\ P(y_n) \prod_i P(f_i|y_n) \end{bmatrix} \quad (17)$$

Ensuite, on somme tout pour avoir la probabilité de l'évidence :

$$P(f_1 \dots f_n) = \sum_i P(y_i) \prod_i P(f_i|y_i) \quad (18)$$

Finalement, on normalise en divisant (17) par (18) et on retrouve $P(Y|f_1 \dots f_n)$.