Super VIP Cheatsheet: Artificial Intelligence

Afshine Amidi and Shervine Amidi

September 8, 2019

Contents

1 Reflex-based models

1.1 Linear predictors

In this section, we will go through reflex-based models that can improve with experience, by going through samples that have input-output pairs.

❒ **Feature vector** – The feature vector of an input x is noted $\phi(x)$ and is such that:

$$\boxed{\phi(x)=\left[\begin{array}{c}\phi_1(x)\\\vdots\\\phi_d(x)\end{array}\right]\in\mathbb{R}^d}$$

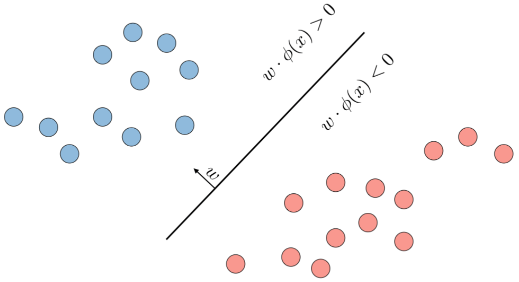
❒ **Score** – The score $s(x,w)$ of an example $(\phi(x),y)\in\mathbb{R}^d\times\mathbb{R}$ associated to a linear model of weights $w\in\mathbb{R}^d$ is given by the inner product:

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

1.1.1 Classification

❒ **Linear classifier** – Given a weight vector $w\in\mathbb{R}^d$ and a feature vector $\phi(x)\in\mathbb{R}^d$, the binary linear classifier f_w is given by:

$$\boxed{f_w(x)=\textrm{sign}(s(x,w))=\left\{\begin{array}{cl}+1 & \textrm{if }w\cdot\phi(x)>0\\-1 & \textrm{if }w\cdot\phi(x)<0\\? & \textrm{if }w\cdot\phi(x)=0\end{array}\right.}$$



❒ **Margin** – The margin $m(x,y,w)\in\mathbb{R}$ of an example $(\phi(x),y)\in\mathbb{R}^d\times\{-1,+1\}$ associated to a linear model of weights $w\in\mathbb{R}^d$ quantifies the confidence of the prediction: larger values are better. It is given by:

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

1.1.2 Regression

❒ **Linear regression** – Given a weight vector $w\in\mathbb{R}^d$ and a feature vector $\phi(x)\in\mathbb{R}^d$, the output of a linear regression of weights w denoted as f_w is given by:

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

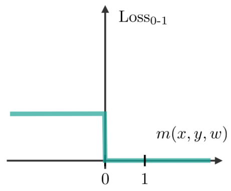
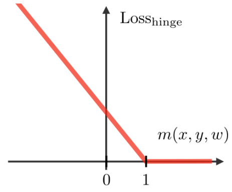
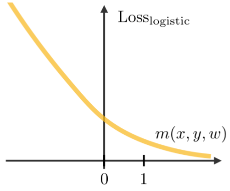
❒ **Residual** – The residual $\textrm{res}(x,y,w)\in\mathbb{R}$ is defined as being the amount by which the prediction $f_w(x)$ overshoots the target y:

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

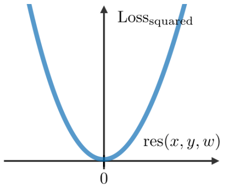
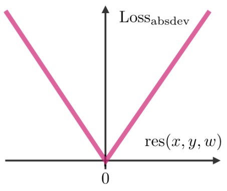
1.2 Loss minimization

❒ **Loss function** – A loss function $\textrm{Loss}(x,y,w)$ quantifies how unhappy we are with the weights w of the model in the prediction task of output y from input x. It is a quantity we want to minimize during the training process.

❒ **Classification case** – The classification of a sample x of true label $y\in\{-1,+1\}$ with a linear model of weights w can be done with the predictor $f_w(x)\triangleq\textrm{sign}(s(x,w))$. In this situation, a metric of interest quantifying the quality of the classification is given by the margin $m(x,y,w)$, and can be used with the following loss functions:

Name	Zero-one loss	Hinge loss	Logistic loss
$\textrm{Loss}(x,y,w)$	$1_{\{m(x,y,w)\leqslant0\}}$	$\max(1-m(x,y,w),0)$	$\log(1+e^{-m(x,y,w)})$
Illustration			

❒ **Regression case** – The prediction of a sample x of true label $y\in\mathbb{R}$ with a linear model of weights w can be done with the predictor $f_w(x)\triangleq s(x,w)$. In this situation, a metric of interest quantifying the quality of the regression is given by the margin $\textrm{res}(x,y,w)$ and can be used with the following loss functions:

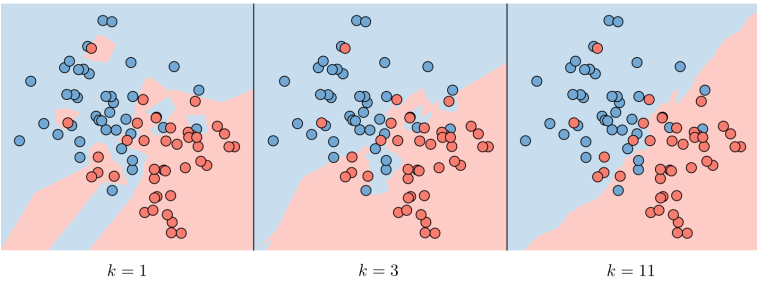
Name	Squared loss	Absolute deviation loss
$\textrm{Loss}(x,y,w)$	$(\textrm{res}(x,y,w))^2$	$\lvert\textrm{res}(x,y,w)\rvert$
Illustration		

❒ **Loss minimization framework** – In order to train a model, we want to minimize the training loss is defined as follows:

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

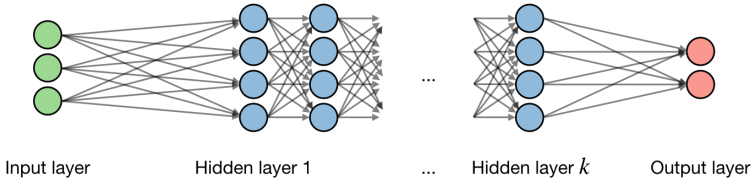
1.3 Non-linear predictors

❒ ***k*-nearest neighbors** – The k-nearest neighbors algorithm, commonly known as k-NN, is a non-parametric approach where the response of a data point is determined by the nature of its k neighbors from the training set. It can be used in both classification and regression settings.



Remark: the higher the parameter k, the higher the bias, and the lower the parameter k, the higher the variance.

❒ **Neural networks** – Neural networks are a class of models that are built with layers. Commonly used types of neural networks include convolutional and recurrent neural networks. The vocabulary around neural networks architectures is described in the figure below:



By noting i the i^{th} layer of the network and j the j^{th} hidden unit of the layer, we have:

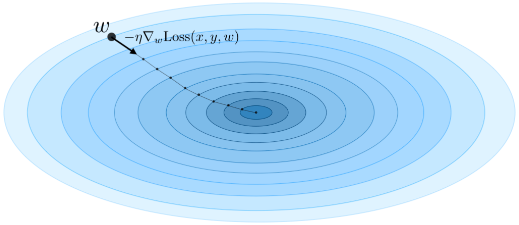
$$\boxed{z_j^{[i]}={w_j^{[i]}}^Tx+b_j^{[i]}}$$

where we note w, b, x, z the weight, bias, input and non-activated output of the neuron respectively.

1.4 Stochastic gradient descent

❒ **Gradient descent** – By noting $\eta\in\mathbb{R}$ the learning rate (also called step size), the update rule for gradient descent is expressed with the learning rate and the loss function $\textrm{Loss}(x,y,w)$ as follows:

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



❒ **Stochastic updates** – Stochastic gradient descent (SGD) updates the parameters of the model one training example $(\phi(x),y)\in\mathcal{D}_{\textrm{train}}$ at a time. This method leads to sometimes noisy, but fast updates.

❒ **Batch updates** – Batch gradient descent (BGD) updates the parameters of the model one batch of examples (e.g. the entire training set) at a time. This method computes stable update directions, at a greater computational cost.

1.5 Fine-tuning models

❒ **Hypothesis class** – A hypothesis class $\mathcal{F}$ is the set of possible predictors with a fixed $\phi(x)$ and varying w:

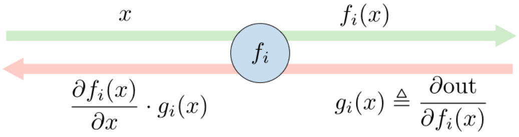
$$\boxed{\mathcal{F}=\left\{f_w:w\in\mathbb{R}^d\right\}}$$

❒ **Logistic function** – The logistic function σ, also called the sigmoid function, is defined as:

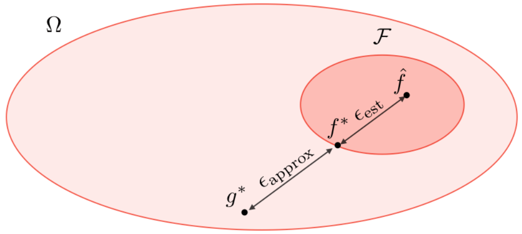
$$\boxed{\forall z\in]-\infty,+\infty[,\quad\sigma(z)=\frac{1}{1+e^{-z}}}$$

Remark: we have $\sigma'(z)=\sigma(z)(1-\sigma(z))$.

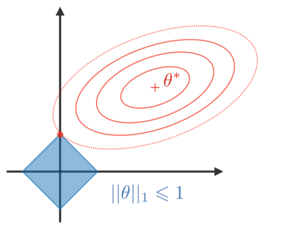
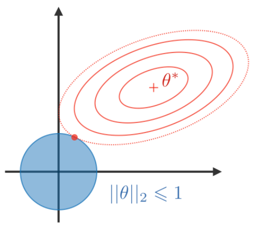
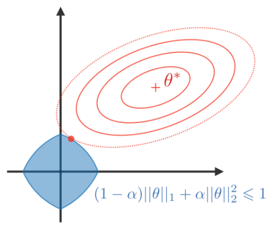
❒ **Backpropagation** – The forward pass is done through f_i, which is the value for the subexpression rooted at i, while the backward pass is done through $g_i=\frac{\partial\textrm{out}}{\partial f_i}$ and represents how f_i influences the output.


x
$f_i(x)$
f_i
$\frac{\partial f_i(x)}{\partial x}\cdot g_i(x)$
$g_i(x)\triangleq\frac{\partial\textrm{out}}{\partial f_i(x)}$

❒ **Approximation and estimation error** – The approximation error $\epsilon_{\textrm{approx}}$ represents how far the entire hypothesis class $\mathcal{F}$ is from the target predictor g^*, while the estimation error $\epsilon_{\textrm{est}}$ quantifies how good the predictor $\hat{f}$ is with respect to the best predictor f^* of the hypothesis class $\mathcal{F}$.



❒ **Regularization** – The regularization procedure aims at avoiding the model to overfit the data and thus deals with high variance issues. The following table sums up the different types of commonly used regularization techniques:

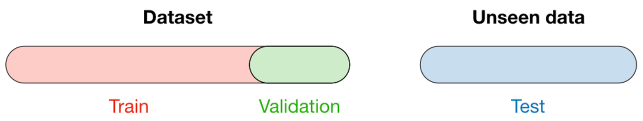
LASSO	Ridge	Elastic Net
- Shrinks coefficients to 0 - Good for variable selection	Makes coefficients smaller	Tradeoff between variable selection and small coefficients
		 $+\theta^*$ $(1-\alpha)\|\|\theta\|\|_1 + \alpha\|\|\theta\|\|_2^2 \leqslant 1$
$... + \lambda\|\|\theta\|\|_1$ $\lambda \in \mathbb{R}$	$... + \lambda\|\|\theta\|\|_2^2$ $\lambda \in \mathbb{R}$	$... + \lambda\Big[(1-\alpha)\|\|\theta\|\|_1 + \alpha\|\|\theta\|\|_2^2\Big]$ $\lambda \in \mathbb{R}, \quad \alpha \in [0,1]$

❒ **Hyperparameters** – Hyperparameters are the properties of the learning algorithm, and include features, regularization parameter λ, number of iterations T, step size η, etc.

❒ **Sets vocabulary** – When selecting a model, we distinguish 3 different parts of the data that we have as follows:

Training set	Validation set	Testing set
- Model is trained - Usually 80 of the dataset	- Model is assessed - Usually 20 of the dataset - Also called hold-out	- Model gives predictions - Unseen data or development set

Once the model has been chosen, it is trained on the entire dataset and tested on the unseen test set. These are represented in the figure below:



1.6 Unsupervised Learning

The class of unsupervised learning methods aims at discovering the structure of the data, which may have of rich latent structures.

1.6.1 k-means

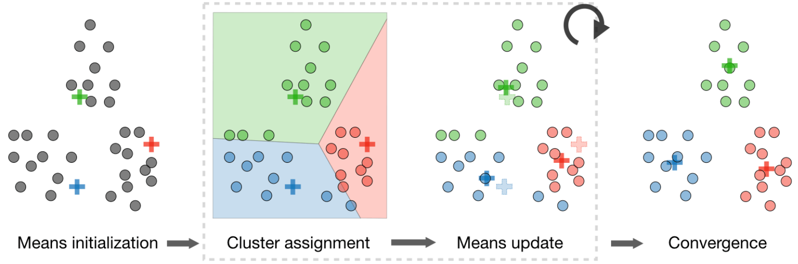
❒ **Clustering** – Given a training set of input points $\mathcal{D}_{\textrm{train}}$, the goal of a clustering algorithm is to assign each point $\phi(x_i)$ to a cluster $z_i \in \{1,...,k\}$.

❒ **Objective function** – The loss function for one of the main clustering algorithms, k-means, is given by:

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

❒ **Algorithm** – After randomly initializing the cluster centroids $\mu_1,\mu_2,...,\mu_k \in \mathbb{R}^n$, the k-means algorithm repeats the following step until convergence:

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



1.6.2 Principal Component Analysis

❒ **Eigenvalue, eigenvector** – Given a matrix $A \in \mathbb{R}^{n\times n}$, λ is said to be an eigenvalue of A if there exists a vector $z \in \mathbb{R}^n\backslash\{0\}$, called eigenvector, such that we have:

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

❐ **Spectral theorem** – Let $A \in \mathbb{R}^{n\times n}$. If A is symmetric, then A is diagonalizable by a real orthogonal matrix $U \in \mathbb{R}^{n\times n}$. By noting $\Lambda = \textrm{diag}(\lambda_1,...,\lambda_n)$, we have:

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

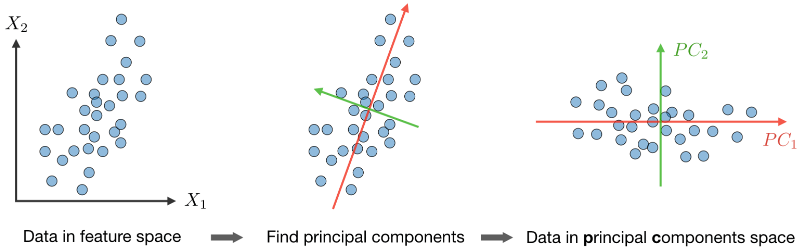
Remark: the eigenvector associated with the largest eigenvalue is called principal eigenvector of matrix A.

❐ **Algorithm** – The Principal Component Analysis (PCA) procedure is a dimension reduction technique that projects the data on k dimensions by maximizing the variance of the data as follows:

- Step 1: Normalize the data to have a mean of 0 and standard deviation of 1.

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

- Step 2: Compute $\Sigma = \displaystyle\frac{1}{m}\sum_{i=1}^m x^{(i)}{x^{(i)}}^T \in \mathbb{R}^{n\times n}$, which is symmetric with real eigenvalues.
- Step 3: Compute $u_1, ..., u_k \in \mathbb{R}^n$ the k orthogonal principal eigenvectors of Σ, i.e. the orthogonal eigenvectors of the k largest eigenvalues.
- Step 4: Project the data on $\textrm{span}_\mathbb{R}(u_1,...,u_k)$. This procedure maximizes the variance among all k-dimensional spaces.



Data in feature space → Find principal components → Data in principal components space

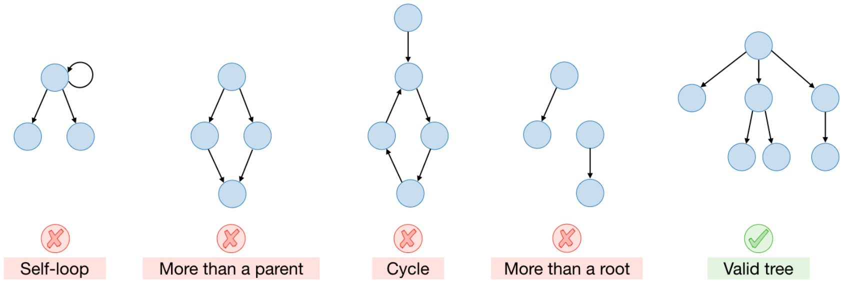
2 States-based models

2.1 Search optimization

In this section, we assume that by accomplishing action a from state s, we deterministically arrive in state Succ(s,a). The goal here is to determine a sequence of actions $(a_1,a_2,a_3,a_4,...)$ that starts from an initial state and leads to an end state. In order to solve this kind of problem, our objective will be to find the minimum cost path by using states-based models.

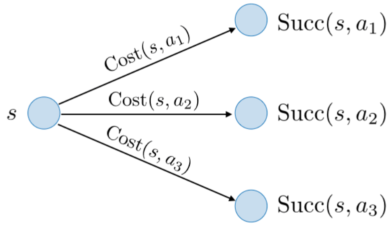
2.1.1 Tree search

This category of states-based algorithms explores all possible states and actions. It is quite memory efficient, and is suitable for huge state spaces but the runtime can become exponential in the worst cases.



❐ **Search problem** – A search problem is defined with:

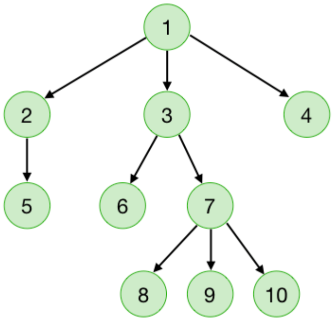
- a starting state $s_{\textrm{start}}$
- possible actions Actions(s) from state s
- action cost Cost(s,a) from state s with action a
- successor Succ(s,a) of state s after action a
- whether an end state was reached IsEnd(s)



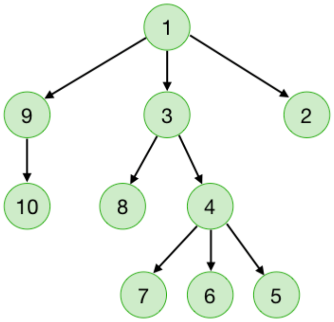
The objective is to find a path that minimizes the cost.

❐ **Backtracking search** – Backtracking search is a naive recursive algorithm that tries all possibilities to find the minimum cost path. Here, action costs can be either positive or negative.

❐ **Breadth-first search (BFS)** – Breadth-first search is a graph search algorithm that does a level-by-level traversal. We can implement it iteratively with the help of a queue that stores at each step future nodes to be visited. For this algorithm, we can assume action costs to be equal to a constant $c \geqslant 0$.



❒ **Depth-first search (DFS)** – Depth-first search is a search algorithm that traverses a graph by following each path as deep as it can. We can implement it recursively, or iteratively with the help of a stack that stores at each step future nodes to be visited. For this algorithm, action costs are assumed to be equal to 0.



❒ **Iterative deepening** – The iterative deepening trick is a modification of the depth-first search algorithm so that it stops after reaching a certain depth, which guarantees optimality when all action costs are equal. Here, we assume that action costs are equal to a constant $c \geqslant 0$.

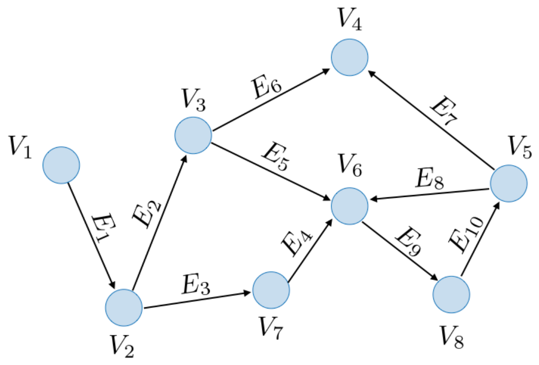
❒ **Tree search algorithms summary** – By noting b the number of actions per state, d the solution depth, and D the maximum depth, we have:

Algorithm	Action costs	Space	Time
Backtracking search	any	$\mathcal{O}(D)$	$\mathcal{O}(b^D)$
Breadth-first search	$c \geqslant 0$	$\mathcal{O}(b^d)$	$\mathcal{O}(b^d)$
Depth-first search	0	$\mathcal{O}(D)$	$\mathcal{O}(b^D)$
DFS-Iterative deepening	$c \geqslant 0$	$\mathcal{O}(d)$	$\mathcal{O}(b^d)$

2.1.2 Graph search

This category of states-based algorithms aims at constructing optimal paths, enabling exponential savings. In this section, we will focus on dynamic programming and uniform cost search.

❒ **Graph** – A graph is comprised of a set of vertices V (also called nodes) as well as a set of edges E (also called links).

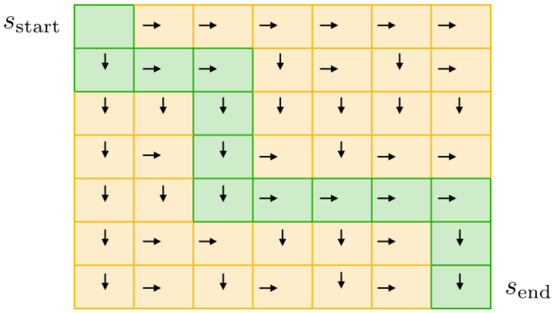


Remark: a graph is said to be acylic when there is no cycle.

❒ **State** – A state is a summary of all past actions sufficient to choose future actions optimally.

❒ **Dynamic programming** – Dynamic programming (DP) is a backtracking search algorithm with memoization (i.e. partial results are saved) whose goal is to find a minimum cost path from state s to an end state s_{end}. It can potentially have exponential savings compared to traditional graph search algorithms, and has the property to only work for acyclic graphs. For any given state s, the future cost is computed as follows:

$$\boxed{\text{FutureCost}(s)=\left\{\begin{array}{lc}0 & \text{if IsEnd}(s)\\ \displaystyle\underset{a\in\text{Actions}(s)}{\text{min}}\Big[\text{Cost}(s,a)+\text{FutureCost}(\text{Succ}(s,a))\Big] & \text{otherwise}\end{array}\right.}$$

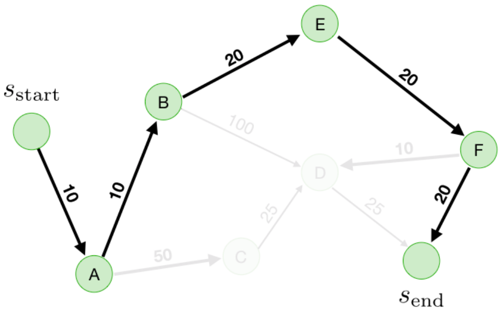


Remark: the figure above illustrates a bottom-to-top approach whereas the formula provides the intuition of a top-to-bottom problem resolution.

❒ **Types of states** – The table below presents the terminology when it comes to states in the context of uniform cost search:

State	Explanation
Explored $\mathcal{E}$	States for which the optimal path has already been found
Frontier $\mathcal{F}$	States seen for which we are still figuring out how to get there with the cheapest cost
Unexplored $\mathcal{U}$	States not seen yet

❐ **Uniform cost search** – Uniform cost search (UCS) is a search algorithm that aims at finding the shortest path from a state $s_{\textrm{start}}$ to an end state $s_{\textrm{end}}$. It explores states s in increasing order of $\textrm{PastCost}(s)$ and relies on the fact that all action costs are non-negative.



Remark 1: the UCS algorithm is logically equivalent to Djikstra's algorithm.
Remark 2: the algorithm would not work for a problem with negative action costs, and adding a positive constant to make them non-negative would not solve the problem since this would end up being a different problem.

❐ **Correctness theorem** – When a state s is popped from the frontier $\mathcal{F}$ and moved to explored set $\mathcal{E}$, its priority is equal to $\textrm{PastCost}(s)$ which is the minimum cost path from $s_{\textrm{start}}$ to s.

❐ **Graph search algorithms summary** – By noting N the number of total states, n of which are explored before the end state $s_{\textrm{end}}$, we have:

Algorithm	Acyclicity	Costs	Time/space
Dynamic programming	yes	any	$\mathcal{O}(N)$
Uniform cost search	no	$c \geqslant 0$	$\mathcal{O}(n\log(n))$

Remark: the complexity countdown supposes the number of possible actions per state to be constant.

2.1.3 Learning costs

Suppose we are not given the values of $\textrm{Cost}(s,a)$, we want to estimate these quantities from a training set of minimizing-cost-path sequence of actions $(a_1, a_2, ..., a_k)$.

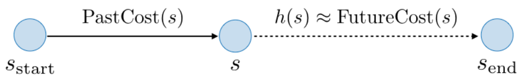
❐ **Structured perceptron** – The structured perceptron is an algorithm aiming at iteratively learning the cost of each state-action pair. At each step, it:

- decreases the estimated cost of each state-action of the true minimizing path y given by the training data,
- increases the estimated cost of each state-action of the current predicted path y' inferred from the learned weights.

Remark: there are several versions of the algorithm, one of which simplifies the problem to only learning the cost of each action a, and the other parametrizes Cost(s,a) to a feature vector of learnable weights.

2.1.4 A* search

❐ **Heuristic function** – A heuristic is a function h over states s, where each $h(s)$ aims at estimating $\textrm{FutureCost}(s)$, the cost of the path from s to $s_{\textrm{end}}$.



❐ **Algorithm** – A^* is a search algorithm that aims at finding the shortest path from a state s to an end state $s_{\textrm{end}}$. It explores states s in increasing order of $\textrm{PastCost}(s) + h(s)$. It is equivalent to a uniform cost search with edge costs $\textrm{Cost}'(s,a)$ given by:

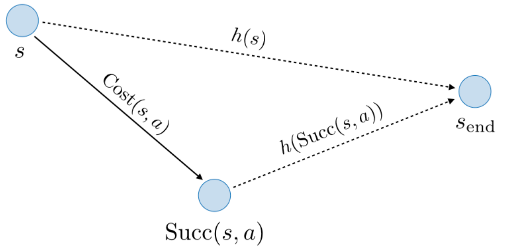
$$\boxed{\textrm{Cost}'(s,a) = \textrm{Cost}(s,a) + h(\textrm{Succ}(s,a)) - h(s)}$$

Remark: this algorithm can be seen as a biased version of UCS exploring states estimated to be closer to the end state.

❐ **Consistency** – A heuristic h is said to be consistent if it satisfies the two following properties:

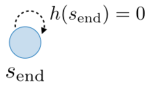
- For all states s and actions a,

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



- The end state verifies the following:

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



❒ **Correctness** – If h is consistent, then A^* returns the minimum cost path.

❒ **Admissibility** – A heuristic h is said to be admissible if we have:

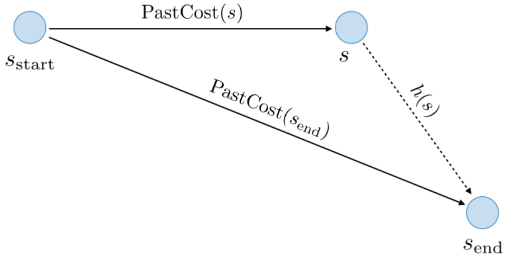
$$\boxed{h(s) \leqslant \textrm{FutureCost}(s)}$$

❒ **Theorem** – Let $h(s)$ be a given heuristic. We have:

$$\boxed{h(s) \textrm{ consistent} \Longrightarrow h(s) \textrm{ admissible}}$$

❒ **Efficiency** – A^* explores all states s satisfying the following equation:

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



Remark: larger values of $h(s)$ is better as this equation shows it will restrict the set of states s going to be explored.

2.1.5 Relaxation

It is a framework for producing consistent heuristics. The idea is to find closed-form reduced costs by removing constraints and use them as heuristics.

❒ **Relaxed search problem** – The relaxation of search problem P with costs Cost is noted $P_{\textrm{rel}}$ with costs $\textrm{Cost}_{\textrm{rel}}$, and satisfies the identity:

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

❒ **Relaxed heuristic** – Given a relaxed search problem $P_{\textrm{rel}}$, we define the relaxed heuristic $h(s) = \textrm{FutureCost}_{\textrm{rel}}(s)$ as the minimum cost path from s to an end state in the graph of costs $\textrm{Cost}_{\textrm{rel}}(s,a)$.

❒ **Consistency of relaxed heuristics** – Let $P_{\textrm{rel}}$ be a given relaxed problem. By theorem, we have:

$$\boxed{h(s) = \textrm{FutureCost}_{\textrm{rel}}(s) \Longrightarrow h(s) \textrm{ consistent}}$$

❒ **Tradeoff when choosing heuristic** – We have to balance two aspects in choosing a heuristic:

- Computational efficiency: $h(s) = \textrm{FutureCost}_{\textrm{rel}}(s)$ must be easy to compute. It has to produce a closed form, easier search and independent subproblems.

- Good enough approximation: the heuristic $h(s)$ should be close to FutureCost(s) and we have thus to not remove too many constraints.

❒ **Max heuristic** – Let $h_1(s)$, $h_2(s)$ be two heuristics. We have the following property:

$$\boxed{h_1(s),\ h_2(s) \textrm{ consistent} \Longrightarrow h(s) = \max\{h_1(s),\ h_2(s)\} \textrm{ consistent}}$$

2.2 Markov decision processes

In this section, we assume that performing action a from state s can lead to several states s_1',s_2',... in a probabilistic manner. In order to find our way between an initial state and an end state, our objective will be to find the maximum value policy by using Markov decision processes that help us cope with randomness and uncertainty.

2.2.1 Notations

❒ **Definition** – The objective of a Markov decision process is to maximize rewards. It is defined with:

- a starting state $s_{\textrm{start}}$

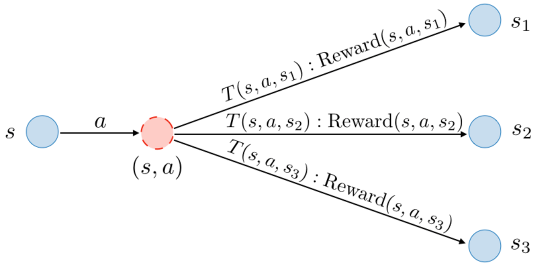
- possible actions Actions(s) from state s

- transition probabilities $T(s,a,s')$ from s to s' with action a

- rewards Reward(s,a,s') from s to s' with action a

- whether an end state was reached IsEnd(s)

- a discount factor $0 \leqslant \gamma \leqslant 1$


s
a
(s,a)
$T(s,a,s_1)$: Reward(s,a,s_1)
$T(s,a,s_2)$: Reward(s,a,s_2)
$T(s,a,s_3)$: Reward(s,a,s_3)
s_1
s_2
s_3

❒ **Transition probabilities** – The transition probability $T(s,a,s')$ specifies the probability of going to state s' after action a is taken in state s. Each $s' \mapsto T(s,a,s')$ is a probability distribution, which means that:

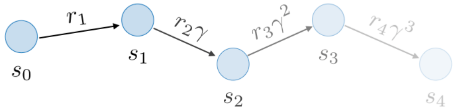
$$\forall s,a, \quad \boxed{\sum_{s' \in \textrm{ States}} T(s,a,s') = 1}$$

❒ **Policy** – A policy π is a function that maps each state s to an action a, i.e.

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

❒ **Utility** – The utility of a path $(s_0, ..., s_k)$ is the discounted sum of the rewards on that path. In other words,

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



Remark: the figure above is an illustration of the case $k=4$.

❒ **Q-value** – The Q-value of a policy π by taking action a from state s, also noted $Q_\pi(s,a)$, is the expected utility of taking action a from state s and then following policy π. It is defined as follows:

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

❒ **Value of a policy** – The value of a policy π from state s, also noted $V_\pi(s)$, is the expected utility by following policy π from state s over random paths. It is defined as follows:

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

Remark: $V_\pi(s)$ is equal to 0 if s is an end state.

2.2.2 Applications

❒ **Policy evaluation** – Given a policy π, policy evaluation is an iterative algorithm that computes V_π. It is done as follows:

- Initialization: for all states s, we have

$$\boxed{V_\pi^{(0)}(s)\longleftarrow0}$$

- Iteration: for t from 1 to $T_{\textrm{PE}}$, we have

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

with

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

Remark: by noting S the number of states, A the number of actions per state, S' the number of successors and T the number of iterations, then the time complexity is of $\mathcal{O}(T_{\textrm{PE}}SS')$.

❒ **Optimal Q-value** – The optimal Q-value $Q_{\textrm{opt}}(s,a)$ of state s with action a is defined to be the maximum Q-value attained by any policy starting. It is computed as follows:

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

❒ **Optimal value** – The optimal value $V_{\textrm{opt}}(s)$ of state s is defined as being the maximum value attained by any policy. It is computed as follows:

$$\boxed{V_{\textrm{opt}}(s)=\underset{a\in\textrm{ Actions}(s)}{\textrm{max}}Q_{\textrm{opt}}(s,a)}$$

❒ **Optimal policy** – The optimal policy $\pi_{\textrm{opt}}$ is defined as being the policy that leads to the optimal values. It is defined by:

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

❒ **Value iteration** – Value iteration is an algorithm that finds the optimal value $V_{\textrm{opt}}$ as well as the optimal policy $\pi_{\textrm{opt}}$. It is done as follows:

- Initialization: for all states s, we have

$$\boxed{V_{\textrm{opt}}^{(0)}(s)\longleftarrow0}$$

- Iteration: for t from 1 to $T_{\textrm{VI}}$, we have

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

with

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

Remark: if we have either $\gamma<1$ or the MDP graph being acyclic, then the value iteration algorithm is guaranteed to converge to the correct answer.

2.2.3 When unknown transitions and rewards

Now, let's assume that the transition probabilities and the rewards are unknown.

❒ **Model-based Monte Carlo** – The model-based Monte Carlo method aims at estimating $T(s,a,s')$ and $\textrm{Reward}(s,a,s')$ using Monte Carlo simulation with:

$$\boxed{\widehat{T}(s,a,s')=\frac{\#\textrm{ times }(s,a,s')\textrm{ occurs}}{\#\textrm{ times }(s,a)\textrm{ occurs}}}$$

and

$$\boxed{\widehat{\textrm{Reward}}(s,a,s')=r\textrm{ in }(s,a,r,s')}$$

These estimations will be then used to deduce Q-values, including Q_π and $Q_{\textrm{opt}}$.

Remark: model-based Monte Carlo is said to be off-policy, because the estimation does not depend on the exact policy.

❐ **Model-free Monte Carlo** – The model-free Monte Carlo method aims at directly estimating Q_π, as follows:

$$\boxed{\widehat{Q}_\pi(s,a)=\textrm{average of }u_t\textrm{ where }s_{t-1}=s, a_t=a}$$

where u_t denotes the utility starting at step t of a given episode.

Remark: model-free Monte Carlo is said to be on-policy, because the estimated value is dependent on the policy π used to generate the data.

❐ **Equivalent formulation** – By introducing the constant $\eta=\frac{1}{1+(\#\textrm{updates to }(s,a))}$ and for each (s,a,u) of the training set, the update rule of model-free Monte Carlo has a convex combination formulation:

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

as well as a stochastic gradient formulation:

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

❐ **SARSA** – State-action-reward-state-action (SARSA) is a boostrapping method estimating Q_π by using both raw data and estimates as part of the update rule. For each (s,a,r,s',a'), we have:

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

Remark: the SARSA estimate is updated on the fly as opposed to the model-free Monte Carlo one where the estimate can only be updated at the end of the episode.

❐ **Q-learning** – Q-learning is an off-policy algorithm that produces an estimate for $Q_{\textrm{opt}}$. On each (s,a,r,s',a'), we have:

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

❐ **Epsilon-greedy** – The epsilon-greedy policy is an algorithm that balances exploration with probability ϵ and exploitation with probability $1-\epsilon$. For a given state s, the policy $\pi_{\textrm{act}}$ is computed as follows:

$$\boxed{\pi_{\textrm{act}}(s)=\left\{\begin{array}{ll}\underset{a\in\textrm{ Actions}}{\textrm{argmax}}\ \widehat{Q}_{\textrm{opt}}(s,a) & \textrm{with proba }1-\epsilon\\\textrm{random from Actions}(s) & \textrm{with proba }\epsilon\end{array}\right.}$$

2.3 Game playing

In games (e.g. chess, backgammon, Go), other agents are present and need to be taken into account when constructing our policy.

❐ **Game tree** – A game tree is a tree that describes the possibilities of a game. In particular, each node is a decision point for a player and each root-to-leaf path is a possible outcome of the game.

❐ **Two-player zero-sum game** – It is a game where each state is fully observed and such that players take turns. It is defined with:

- a starting state $s_{\textrm{start}}$
- possible actions Actions(s) from state s
- successors Succ(s,a) from states s with actions a
- whether an end state was reached IsEnd(s)
- the agent's utility Utility(s) at end state s
- the player Player(s) who controls state s

Remark: we will assume that the utility of the agent has the opposite sign of the one of the opponent.

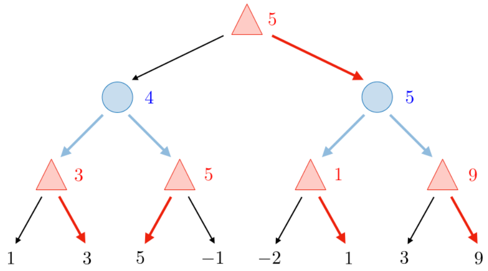
❐ **Types of policies** – There are two types of policies:

- Deterministic policies, noted $\pi_p(s)$, which are actions that player p takes in state s.
- Stochastic policies, noted $\pi_p(s,a)\in[0,1]$, which are probabilities that player p takes action a in state s.

❐ **Expectimax** – For a given state s, the expectimax value $V_{\textrm{exptmax}}(s)$ is the maximum expected utility of any agent policy when playing with respect to a fixed and known opponent policy $\pi_{\textrm{opp}}$. It is computed as follows:

$$\boxed{V_{\textrm{exptmax}}(s)=\left\{\begin{array}{ll}\textrm{Utility}(s) & \textrm{IsEnd}(s)\\\underset{a\in\textrm{Actions}(s)}{\textrm{max}}V_{\textrm{exptmax}}(\textrm{Succ}(s,a)) & \textrm{Player}(s)=\textrm{agent}\\\displaystyle\sum_{a\in\textrm{Actions}(s)}\pi_{\textrm{opp}}(s,a)V_{\textrm{exptmax}}(\textrm{Succ}(s,a)) & \textrm{Player}(s)=\textrm{opp}\end{array}\right.}$$

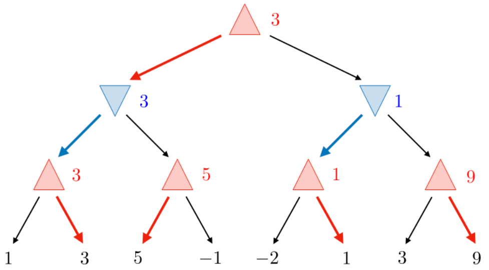
Remark: expectimax is the analog of value iteration for MDPs.



❐ **Minimax** – The goal of minimax policies is to find an optimal policy against an adversary by assuming the worst case, i.e. that the opponent is doing everything to minimize the agent's utility. It is done as follows:

$$\boxed{V_{\textrm{minimax}}(s)=\left\{\begin{array}{ll}\textrm{Utility}(s) & \textrm{IsEnd}(s)\\\underset{a\in\textrm{Actions}(s)}{\textrm{max}}V_{\textrm{minimax}}(\textrm{Succ}(s,a)) & \textrm{Player}(s)=\textrm{agent}\\\underset{a\in\textrm{Actions}(s)}{\textrm{min}}V_{\textrm{minimax}}(\textrm{Succ}(s,a)) & \textrm{Player}(s)=\textrm{opp}\end{array}\right.}$$

Remark: we can extract π_{max} and π_{min} from the minimax value $V_{minimax}$.



❒ **Minimax properties** – By noting V the value function, there are 3 properties around minimax to have in mind:

- *Property 1*: if the agent were to change its policy to any $\pi_{\textrm{agent}}$, then the agent would be no better off.

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

- *Property 2*: if the opponent changes its policy from $\pi_{\textrm{min}}$ to $\pi_{\textrm{opp}}$, then he will be no better off.

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

- *Property 3*: if the opponent is known to be not playing the adversarial policy, then the minimax policy might not be optimal for the agent.

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

In the end, we have the following relationship:

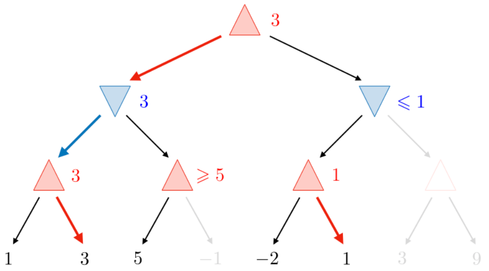
$$\boxed{V(\pi_{\textrm{exptmax}},\pi_{\min})\leqslant V(\pi_{\max},\pi_{\min})\leqslant V(\pi_{\max},\pi)\leqslant V(\pi_{\textrm{exptmax}},\pi)}$$

2.3.1 Speeding up minimax

❒ **Evaluation function** – An evaluation function is a domain-specific and approximate estimate of the value $V_{\textrm{minimax}}(s)$. It is noted $\textrm{Eval}(s)$.

Remark: FutureCost(s) is an analogy for search problems.

❒ **Alpha-beta pruning** – Alpha-beta pruning is a domain-general exact method optimizing the minimax algorithm by avoiding the unnecessary exploration of parts of the game tree. To do so, each player keeps track of the best value they can hope for (stored in α for the maximizing player and in β for the minimizing player). At a given step, the condition $\beta<\alpha$ means that the optimal path is not going to be in the current branch as the earlier player had a better option at their disposal.



❒ **TD learning** – Temporal difference (TD) learning is used when we don't know the transitions/rewards. The value is based on exploration policy. To be able to use it, we need to know rules of the game $\textrm{Succ}(s,a)$. For each (s,a,r,s'), the update is done as follows:

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

2.3.2 Simultaneous games

This is the contrary of turn-based games, where there is no ordering on the player's moves.

❒ **Single-move simultaneous game** – Let there be two players A and B, with given possible actions. We note $V(a,b)$ to be A's utility if A chooses action a, B chooses action b. V is called the payoff matrix.

❒ **Strategies** – There are two main types of strategies:

- A pure strategy is a single action:

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

- A mixed strategy is a probability distribution over actions:

$$\forall a\in\textrm{Actions},\quad\boxed{0\leqslant\pi(a)\leqslant1}$$

❒ **Game evaluation** – The value of the game $V(\pi_A,\pi_B)$ when player A follows π_A and player B follows π_B is such that:

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

❒ **Minimax theorem** – By noting π_A,π_B ranging over mixed strategies, for every simultaneous two-player zero-sum game with a finite number of actions, we have:

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

2.3.3 Non-zero-sum games

❒ **Payoff matrix** – We define $V_p(\pi_A,\pi_B)$ to be the utility for player p.

❒ **Nash equilibrium** – A Nash equilibrium is (π_A^*,π_B^*) such that no player has an incentive to change its strategy. We have:

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

Remark: in any finite-player game with finite number of actions, there exists at least one Nash equilibrium.

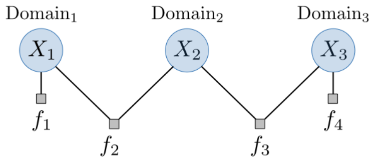
3 Variables-based models

3.1 Constraint satisfaction problems

In this section, our objective is to find maximum weight assignments of variable-based models. One advantage compared to states-based models is that these algorithms are more convenient to encode problem-specific constraints.

3.1.1 Factor graphs

❒ **Definition** – A factor graph, also referred to as a Markov random field, is a set of variables $X=(X_1,...,X_n)$ where $X_i\in\textrm{Domain}_i$ and m factors $f_1,...,f_m$ with each $f_j(X)\geqslant 0$.



❒ **Scope and arity** – The scope of a factor f_j is the set of variables it depends on. The size of this set is called the arity.

Remark: factors of arity 1 and 2 are called unary and binary respectively.

❒ **Assignment weight** – Each assignment $x=(x_1,...,x_n)$ yields a weight $\textrm{Weight}(x)$ defined as being the product of all factors f_j applied to that assignment. Its expression is given by:

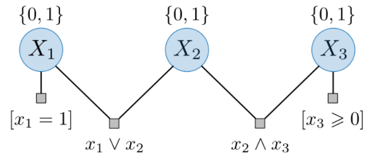
$$\boxed{\textrm{Weight}(x)=\prod_{j=1}^m f_j(x)}$$

❒ **Constraint satisfaction problem** – A constraint satisfaction problem (CSP) is a factor graph where all factors are binary; we call them to be constraints:

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

Here, the constraint j with assignment x is said to be satisfied if and only if $f_j(x)=1$.

❒ **Consistent assignment** – An assignment x of a CSP is said to be consistent if and only if $\textrm{Weight}(x)=1$, i.e. all constraints are satisfied.



3.1.2 Dynamic ordering

❒ **Dependent factors** – The set of dependent factors of variable X_i with partial assignment x is called $D(x,X_i)$, and denotes the set of factors that link X_i to already assigned variables.

❐ **Backtracking search** – Backtracking search is an algorithm used to find maximum weight assignments of a factor graph. At each step, it chooses an unassigned variable and explores its values by recursion. Dynamic ordering (*i.e.* choice of variables and values) and lookahead (*i.e.* early elimination of inconsistent options) can be used to explore the graph more efficiently, although the worst-case runtime stays exponential: $O(|\text{Domain}|^n)$.

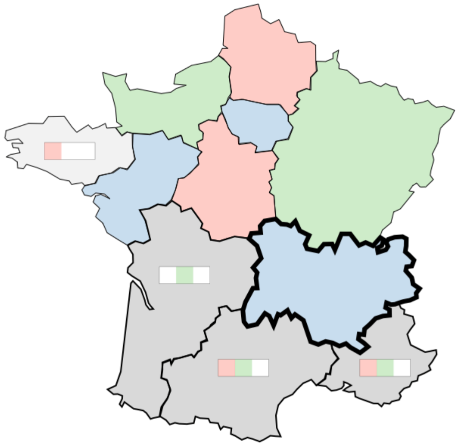
❐ **Forward checking** – It is a one-step lookahead heuristic that preemptively removes inconsistent values from the domains of neighboring variables. It has the following characteristics:

- After assigning a variable X_i, it eliminates inconsistent values from the domains of all its neighbors.
- If any of these domains becomes empty, we stop the local backtracking search.
- If we un-assign a variable X_i, we have to restore the domain of its neighbors.

❐ **Most constrained variable** – It is a variable-level ordering heuristic that selects the next unassigned variable that has the fewest consistent values. This has the effect of making inconsistent assignments to fail earlier in the search, which enables more efficient pruning.

❐ **Least constrained value** – It is a value-level ordering heuristic that assigns the next value that yields the highest number of consistent values of neighboring variables. Intuitively, this procedure chooses first the values that are most likely to work.

Remark: in practice, this heuristic is useful when all factors are constraints.



The example above is an illustration of the 3-color problem with backtracking search coupled with most constrained variable exploration and least constrained value heuristic, as well as forward checking at each step.

❐ **Arc consistency** – We say that arc consistency of variable X_l with respect to X_k is enforced when for each $x_l \in \text{Domain}_l$:

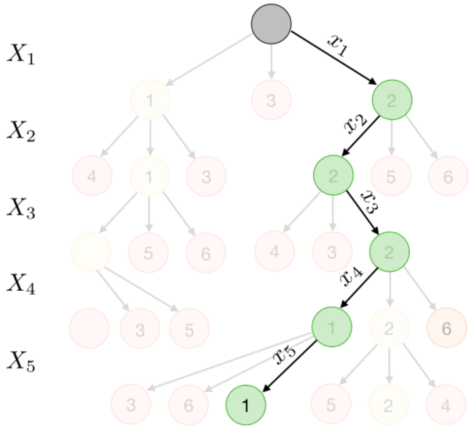
- unary factors of X_l are non-zero,
- there exists at least one $x_k \in \text{Domain}_k$ such that any factor between X_l and X_k is non-zero.

❐ **AC-3** – The AC-3 algorithm is a multi-step lookahead heuristic that applies forward checking to all relevant variables. After a given assignment, it performs forward checking and then successively enforces arc consistency with respect to the neighbors of variables for which the domain change during the process.

Remark: AC-3 can be implemented both iteratively and recursively.

3.1.3 Approximate methods

❐ **Beam search** – Beam search is an approximate algorithm that extends partial assignments of n variables of branching factor $b = |\text{Domain}|$ by exploring the K top paths at each step. The beam size $K \in \{1,...,b^n\}$ controls the tradeoff between efficiency and accuracy. This algorithm has a time complexity of $O(n \cdot Kb\log(Kb))$.
The example below illustrates a possible beam search of parameters $K = 2$, $b = 3$ and $n = 5$.



Remark: $K = 1$ corresponds to greedy search whereas $K \to +\infty$ is equivalent to BFS tree search.

❐ **Iterated conditional modes** – Iterated conditional modes (ICM) is an iterative approximate algorithm that modifies the assignment of a factor graph one variable at a time until convergence. At step i, we assign to X_i the value v that maximizes the product of all factors connected to that variable.

Remark: ICM may get stuck in local minima.

❐ **Gibbs sampling** – Gibbs sampling is an iterative approximate method that modifies the assignment of a factor graph one variable at a time until convergence. At step i:

- we assign to each element $u \in \text{Domain}_i$ a weight $w(u)$ that is the product of all factors connected to that variable,
- we sample v from the probability distribution induced by w and assign it to X_i.

Remark: Gibbs sampling can be seen as the probabilistic counterpart of ICM. It has the advantage to be able to escape local minima in most cases.

3.1.4 Factor graph transformations

❐ **Independence** – Let A,B be a partitioning of the variables X. We say that A and B are independent if there are no edges between A and B and we write:

$$\boxed{A,B \text{ independent} \iff A \perp\!\!\!\perp B}$$

Remark: independence is the key property that allows us to solve subproblems in parallel.

❐ **Conditional independence** – We say that A and B are conditionally independent given C if conditioning on C produces a graph in which A and B are independent. In this case, it is written:

$$\boxed{A \text{ and } B \text{ cond. indep. given } C \Longleftrightarrow A \perp\!\!\!\perp B|C}$$

❒ **Conditioning** – Conditioning is a transformation aiming at making variables independent that breaks up a factor graph into smaller pieces that can be solved in parallel and can use backtracking. In order to condition on a variable $X_i = v$, we do as follows:

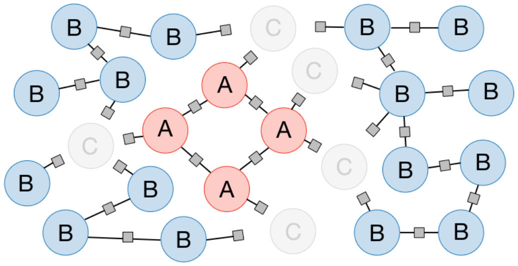
- Consider all factors $f_1,...,f_k$ that depend on X_i
- Remove X_i and $f_1,...,f_k$
- Add $g_j(x)$ for $j \in \{1,...,k\}$ defined as:

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

❒ **Markov blanket** – Let $A \subseteq X$ be a subset of variables. We define MarkovBlanket(A) to be the neighbors of A that are not in A.

❒ **Proposition** – Let $C = \text{MarkovBlanket}(A)$ and $B = X\backslash(A \cup C)$. Then we have:

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



❒ **Elimination** – Elimination is a factor graph transformation that removes X_i from the graph and solves a small subproblem conditioned on its Markov blanket as follows:

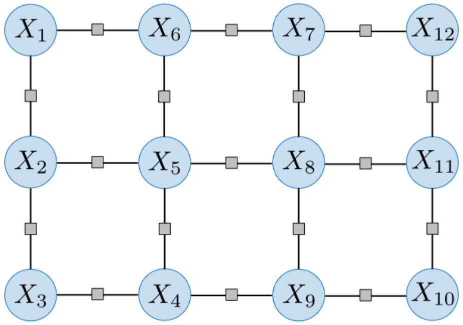
- Consider all factors $f_{i,1},...,f_{i,k}$ that depend on X_i
- Remove X_i and $f_{i,1},...,f_{i,k}$
- Add $f_{\text{new},i}(x)$ defined as:

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

❒ **Treewidth** – The treewidth of a factor graph is the maximum arity of any factor created by variable elimination with the best variable ordering. In other words,

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

The example below illustrates the case of a factor graph of treewidth 3.



Remark: finding the best variable ordering is a NP-hard problem.

3.2 Bayesian networks

In this section, our goal will be to compute conditional probabilities. What is the probability of a query given evidence?

3.2.1 Introduction

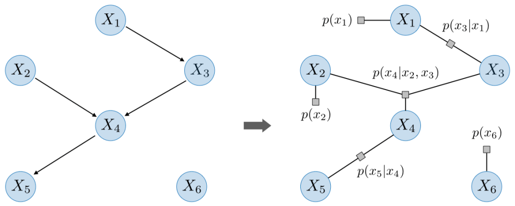
❒ **Explaining away** – Suppose causes C_1 and C_2 influence an effect E. Conditioning on the effect E and on one of the causes (say C_1) changes the probability of the other cause (say C_2). In this case, we say that C_1 has explained away C_2.

❒ **Directed acyclic graph** – A directed acyclic graph (DAG) is a finite directed graph with no directed cycles.

❒ **Bayesian network** – A Bayesian network is a directed acyclic graph (DAG) that specifies a joint distribution over random variables $X = (X_1,...,X_n)$ as a product of local conditional distributions, one for each node:

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

Remark: Bayesian networks are factor graphs imbued with the language of probability.



❒ **Locally normalized** – For each $x_{\text{Parents}(i)}$, all factors are local conditional distributions. Hence they have to satisfy:

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

As a result, sub-Bayesian networks and conditional distributions are consistent.

Remark: local conditional distributions are the true conditional distributions.

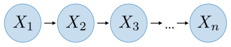
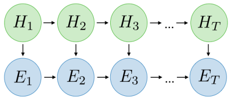
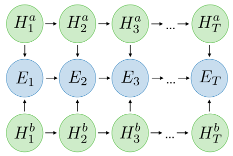
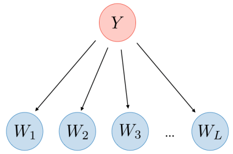
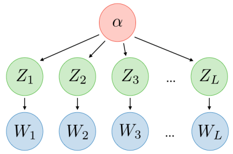
❒ **Marginalization** – The marginalization of a leaf node yields a Bayesian network without that node.

3.2.2 Probabilistic programs

❒ **Concept** – A probabilistic program randomizes variables assignment. That way, we can write down complex Bayesian networks that generate assignments without us having to explicitly specify associated probabilities.

Remark: examples of probabilistic programs include Hidden Markov model (HMM), factorial HMM, naive Bayes, latent Dirichlet allocation, diseases and symptoms and stochastic block models.

❒ **Summary** – The table below summarizes the common probabilistic programs as well as their applications:

Program	Algorithm	Illustration	Example
Markov Model	$X_i \sim p(X_i\|X_{i-1})$		Language modeling
Hidden Markov Model (HMM)	$H_t \sim p(H_t\|H_{t-1})$ $E_t \sim p(E_t\|H_t)$	 H_1 H_2 H_3 ... H_T E_1 E_2 E_3 ... E_T	Object tracking
Factorial HMM	$H_t^o \underset{o\in\{a,b\}}{\sim} p(H_t^o\|H_{t-1}^o)$ $E_t \sim p(E_t\|H_t^a,H_t^b)$	 H_1^a H_2^a H_3^a ... H_T^a E_1 E_2 E_3 ... E_T H_1^b H_2^b H_3^b ... H_T^b	Multiple object tracking
Naive Bayes	$Y \sim p(Y)$ $W_i \sim p(W_i\|Y)$		Document classification
Latent Dirichlet Allocation (LDA)	$\alpha \in \mathbb{R}^K$ distribution $Z_i \sim p(Z_i\|\alpha)$ $W_i \sim p(W_i\|Z_i)$	 α Z_1 Z_2 Z_3 ... Z_L W_1 W_2 W_3 ... W_L	Topic modeling

3.2.3 Inference

❒ **General probabilistic inference strategy** – The strategy to compute the probability $P(Q|E=e)$ of query Q given evidence $E=e$ is as follows:

- Step 1: Remove variables that are not ancestors of the query Q or the evidence E by marginalization
- Step 2: Convert Bayesian network to factor graph
- Step 3: Condition on the evidence $E=e$
- Step 4: Remove nodes disconnected from the query Q by marginalization
- Step 5: Run probabilistic inference algorithm (manual, variable elimination, Gibbs sampling, particle filtering)

❒ **Forward-backward algorithm** – This algorithm computes the exact value of $P(H=h_k|E=e)$ (smoothing query) for any $k\in\{1,...,L\}$ in the case of an HMM of size L. To do so, we proceed in 3 steps:

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

with the convention $F_0 = B_{L+1} = 1$. From this procedure and these notations, we get that

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

Remark: this algorithm interprets each assignment to be a path where each edge $h_{i-1} \to h_i$ is of weight $p(h_i|h_{i-1})p(e_i|h_i)$.

❒ **Gibbs sampling** – This algorithm is an iterative approximate method that uses a small set of assignments (particles) to represent a large probability distribution. From a random assignment x, Gibbs sampling performs the following steps for $i \in \{1,...,n\}$ until convergence:

- For all $u \in \text{Domain}_i$, compute the weight $w(u)$ of assignment x where $X_i = u$
- Sample v from the probability distribution induced by w: $v \sim P(X_i = v|X_{-i} = x_{-i})$
- Set $X_i = v$

Remark: X_{-i} denotes $X\backslash\{X_i\}$ and x_{-i} represents the corresponding assignment.

❒ **Particle filtering** – This algorithm approximates the posterior density of state variables given the evidence of observation variables by keeping track of K particles at a time. Starting from a set of particles C of size K, we run the following 3 steps iteratively:

- Step 1: proposal - For each old particle $x_{t-1} \in C$, sample x from the transition probability distribution $p(x|x_{t-1})$ and add x to a set C'.
- Step 2: weighting - Weigh each x of the set C' by $w(x) = p(e_t|x)$, where e_t is the evidence observed at time t.
- Step 3: resampling - Sample K elements from the set C' using the probability distribution induced by w and store them in C: these are the current particles x_t.

Remark: a more expensive version of this algorithm also keeps track of past particles in the proposal step.

❒ **Maximum likelihood** – If we don't know the local conditional distributions, we can learn them using maximum likelihood.

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

❒ **Laplace smoothing** – For each distribution d and partial assignment $(x_{\text{Parents}(i)},x_i)$, add λ to $\text{count}_d(x_{\text{Parents}(i)},x_i)$, then normalize to get probability estimates.

❒ **Algorithm** – The Expectation-Maximization (EM) algorithm gives an efficient method at estimating the parameter θ through maximum likelihood estimation by repeatedly constructing a lower-bound on the likelihood (E-step) and optimizing that lower bound (M-step) as follows:

- E-step: Evaluate the posterior probability $q(h)$ that each data point e came from a particular cluster h as follows:

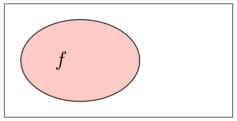
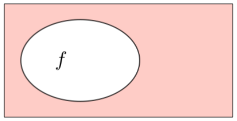
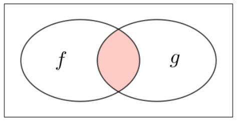
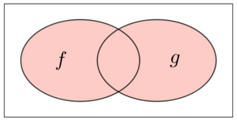
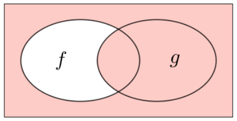
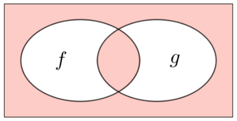
$$\boxed{q(h) = P(H = h|E = e; \theta)}$$

- M-step: Use the posterior probabilities $q(h)$ as cluster specific weights on data points e to determine θ through maximum likelihood.

4 Logic-based models

4.1 Basics

❒ **Syntax of propositional logic** – By noting f,g formulas, and $\neg, \wedge, \vee, \to, \leftrightarrow$ connectives, we can write the following logical expressions:

Name	**Symbol**	**Meaning**	**Illustration**
Affirmation	f	f	
Negation	$\neg f$	not f	
Conjunction	$f \wedge g$	f and g	
Disjunction	$f \vee g$	f or g	
Implication	$f \to g$	if f then g	
Biconditional	$f \leftrightarrow g$	f, that is to say g	

Remark: formulas can be built up recursively out of these connectives.

❒ **Model** – A model w denotes an assignment of binary weights to propositional symbols.

Example: the set of truth values $w = \{A : 0,B : 1,C : 0\}$ is one possible model to the propositional symbols A, B and C.

❒ **Interpretation function** – The interpretation function $\mathcal{I}(f,w)$ outputs whether model w satisfies formula f:

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

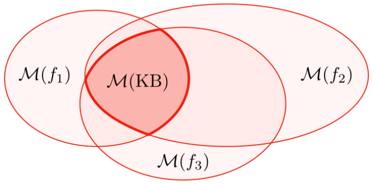
❒ **Set of models** – $\mathcal{M}(f)$ denotes the set of models w that satisfy formula f. Mathematically speaking, we define it as follows:

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

4.2 Knowledge base

❐ **Definition** – The knowledge base KB is the conjunction of all formulas that have been considered so far. The set of models of the knowledge base is the intersection of the set of models that satisfy each formula. In other words:

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



❐ **Probabilistic interpretation** – The probability that query f is evaluated to 1 can be seen as the proportion of models w of the knowledge base KB that satisfy f, i.e.:

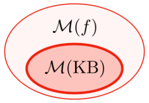
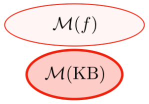
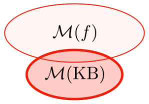
$$\boxed{P(f|\text{KB})=\frac{\displaystyle\sum_{w\in\mathcal{M}(\text{KB})\cap\mathcal{M}(f)}P(W=w)}{\displaystyle\sum_{w\in\mathcal{M}(\text{KB})}P(W=w)}}$$

❐ **Satisfiability** – The knowledge base KB is said to be satisfiable if at least one model w satisfies all its constraints. In other words:

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

Remark: $\mathcal{M}(KB)$ denotes the set of models compatible with all the constraints of the knowledge base.

❐ **Relation between formulas and knowledge base** – We define the following properties between the knowledge base KB and a new formula f:

Name	Mathematical formulation	Illustration	Notes
KB entails f	$\mathcal{M}(\text{KB})\cap\mathcal{M}(f)=\mathcal{M}(\text{KB})$		- f does not bring any new information - Also written KB $\models f$
KB contradicts f	$\mathcal{M}(\text{KB})\cap\mathcal{M}(f)=\varnothing$		- No model satisfies the constraints after adding f Equivalent to KB $\models\neg f$
f contingent to KB	$\mathcal{M}(\text{KB})\cap\mathcal{M}(f)\neq\varnothing$ and $\mathcal{M}(\text{KB})\cap\mathcal{M}(f)\neq\mathcal{M}(\text{KB})$		- f does not contradict KB - f adds a non-trivial amount of information to KB

❐ **Model checking** – A model checking algorithm takes as input a knowledge base KB and outputs whether it is satisfiable or not.

Remark: popular model checking algorithms include DPLL and WalkSat.

❐ **Inference rule** – An inference rule of premises $f_1,...,f_k$ and conclusion g is written:

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

❐ **Forward inference algorithm** – From a set of inference rules Rules, this algorithm goes through all possible $f_1,...,f_k$ and adds g to the knowledge base KB if a matching rule exists. This process is repeated until no more additions can be made to KB.

❐ **Derivation** – We say that KB derives f (written KB $\vdash f$) with rules Rules if f already is in KB or gets added during the forward inference algorithm using the set of rules Rules.

❐ **Properties of inference rules** – A set of inference rules Rules can have the following properties:

Name	Mathematical formulation	Notes
Soundness	$\{f:\text{KB}\vdash f\}\subseteq\{f:\text{KB}\models f\}$	- Inferred formulas are entailed by KB - Can be checked one rule at a time - "*Nothing but the truth*"
Completeness	$\{f:\text{KB}\vdash f\}\supseteq\{f:\text{KB}\models f\}$	- Formulas entailing KB are either already in the knowledge base or inferred from it - "*The whole truth*"

4.3 Propositional logic

In this section, we will go through logic-based models that use logical formulas and inference rules. The idea here is to balance expressivity and computational efficiency.

❒ **Horn clause** – By noting $p_1,...,p_k$ and q propositional symbols, a Horn clause has the form:

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

Remark: when $q = false$, it is called a "goal clause", otherwise we denote it as a "definite clause".

❒ **Modus ponens inference rule** – For propositional symbols $f_1,...,f_k$ and p, the modus ponens rule is written:

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

Remark: it takes linear time to apply this rule, as each application generate a clause that contains a single propositional symbol.

❒ **Completeness** – Modus ponens is complete with respect to Horn clauses if we suppose that KB contains only Horn clauses and p is an entailed propositional symbol. Applying modus ponens will then derive p.

❒ **Conjunctive normal form** – A conjunctive normal form (CNF) formula is a conjunction of clauses, where each clause is a disjunction of atomic formulas.

Remark: in other words, CNFs are $\wedge$ of $\vee$.

❒ **Equivalent representation** – Every formula in propositional logic can be written into an equivalent CNF formula. The table below presents general conversion properties:

Rule name		**Initial**	**Converted**
Eliminate	$\leftrightarrow$	$f\leftrightarrow g$	$(f\rightarrow g)\wedge(g\rightarrow f)$
	$\rightarrow$	$f\rightarrow g$	$\neg f\vee g$
	$\neg\neg$	$\neg\neg f$	f
Distribute	$\neg$ over $\wedge$	$\neg(f\wedge g)$	$\neg f\vee\neg g$
	$\neg$ over $\vee$	$\neg(f\vee g)$	$\neg f\wedge\neg g$
	$\vee$ over $\wedge$	$f\vee(g\wedge h)$	$(f\vee g)\wedge(f\vee h)$

❒ **Resolution inference rule** – For propositional symbols $f_1,...,f_n$, and $g_1,...,g_m$ as well as p, the resolution rule is written:

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

Remark: it can take exponential time to apply this rule, as each application generates a clause that has a subset of the propositional symbols.

❒ **Resolution-based inference** – The resolution-based inference algorithm follows the following steps:

- Step 1: Convert all formulas into CNF
- Step 2: Repeatedly apply resolution rule
- Step 3: Return unsatisfiable if and only if False is derived

4.4 First-order logic

The idea here is that variables yield compact knowledge representations.

❒ **Model** – A model w in first-order logic maps:

- constant symbols to objects
- predicate symbols to tuple of objects

❒ **Horn clause** – By noting $x_1,...,x_n$ variables and $a_1,...,a_k,b$ atomic formulas, the first-order logic version of a horn clause has the form:

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

❒ **Substitution** – A substitution θ maps variables to terms and Subst(θ,f) denotes the result of substitution θ on f.

❒ **Unification** – Unification takes two formulas f and g and returns the most general substitution θ that makes them equal:

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

Note: Unify$[f,g]$ returns Fail if no such θ exists.

❒ **Modus ponens** – By noting $x_1,...,x_n$ variables, $a_1,...,a_k$ and $a'_1,...,a'_k$ atomic formulas and by calling $\theta=\text{Unify}(a'_1\wedge...\wedge a'_k,a_1\wedge...\wedge a_k)$ the first-order logic version of modus ponens can be written:

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

❒ **Completeness** – Modus ponens is complete for first-order logic with only Horn clauses.

❒ **Resolution rule** – By noting $f_1,...,f_n$, $g_1,...,g_m$, p, q formulas and by calling $\theta=\text{Unify}(p,q)$, the first-order logic version of the resolution rule can be written:

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

❒ **Semi-decidability** – First-order logic, even restricted to only Horn clauses, is semi-decidable.

- if KB $\models f$, forward inference on complete inference rules will prove f in finite time
- if KB $\not\models f$, no algorithm can show this in finite time