Linear Predictors

Feature vector - The feature vector of an input x is noted $\phi(x)$ and is such that: $\phi(x) = [\ \phi_1(x) \dots \phi_d(x)\ ] \in \mathbb{R}^d$

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: $s(x, w) = w \cdot \phi(x)$

Classifier

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: $f_w(x) = \text{sign}(s(x, w))$

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: $m(x, y, w) = s(x, w) \times y$

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: $f_w(x) = s(x, w)$

Residual - The residual $\text{res}(x, y, w) \in \mathbb{R}$ is defined as being the amount by which the Prediction $f_w(x)$ overshoots the target y: $\text{res}(x, y, w) = f_w(x) - y$

Loss Minimization

Loss function - A loss function $\text{Loss}(x, y, w)$ 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 $f_w(x) \triangleq \text{sign}(s(x, w))$. 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:

Zero-one loss	$1_{\{m(x,y,w) \leqslant 0\}}$
Hinge loss	$\max(1 - m(x, y, w), 0)$
Logistic loss	$\log(1 + e^{-m(x,y,w)})$
Squared loss	$(\text{res}(x, y, w))^2$
Absolute deviation loss	$\lvert\text{res}(x, y, w)\rvert$

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 $\text{res}(x, y, w)$ and can be used with the following loss functions:

Loss minimization framework - In order to train a model, we want to minimize the training loss is defined as follows: $\text{TrainLoss}(w) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, w)$

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.

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

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

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

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 $\text{Loss}(x, y, w)$ as follows:

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

Stochastic updates - Stochastic gradient descent (SGD) updates the parameters of the model one training example $(\phi(x), y) \in \mathcal{D}_{\text{train}}$ 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.

Fine-tuning models Hypothesis class - A hypothesis class $\mathcal{F}$ is the set of possible Predictors with a fixed $\phi(x)$ and varying w: $\mathcal{F} = \left\{ f_w : w \in \mathbb{R}^d \right\}$

Logistic function - The logistic function σ, also called the sigmoid function, is defined as:

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

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 $g_i = \partial \text{out}/\partial f_i$ and represents how f_i influences the output.

Approximation and estimation error - The approximation error ϵ_{approx} represents how far the entire hypothesis class $\mathcal{F}$ is from the target Predictor $g^\star$, while the estimation error ϵ_{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:

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

Unsupervised Learning

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

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

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

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

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

Search Optimization

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

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

Graph Search

Algorithm	BS	BFS	DFS	DFS-ID
Action costs	any	$c \geqslant 0$	0	$c \geq 0$
Space	$\mathcal{O}(D)$	$\mathcal{O}(b^d)$	$\mathcal{O}(D)$	$\mathcal{O}(d)$
Time	$\mathcal{O}(b^D)$	$\mathcal{O}(b^d)$	$\mathcal{O}(b^D)$	$\mathcal{O}(b^d)$

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:

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

Uniform cost search - Uniform cost search (UCS) is a search algorithm that aims at finding the shortest path from a state s_{start} to an end state s_{end}. It explores states s in increasing order of $\text{PastCost}(s)$ 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 $\text{PastCost}(s)$ which is the minimum cost path from s_{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_{end}, we have:

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

Learning costs

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

Structured perceptron - The structured perceptron is an algorithm aiming at iteratively learning the cost of each state-action pair. At each step, it:
(1) decreases the estimated cost of each state-action of the true minimizing path y given by the training data, (2) 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.

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

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

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

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: (1) For all states s and actions a, $h(s) \leqslant \text{Cost}(s, a) + h(\text{Succ}(s, a))$ (2) The end state verifies the following: $h(s_{\text{end}}) = 0$

Correctness - If h is consistent, then A* returns the minimum cost path.

Admissibility - A heuristic h is said to be admissible if we have: $h(s) \leqslant \text{FutureCost}(s)$

Theorem - Let $h(s)$ be a given heuristic. We have: $h(s)$ consistent $\Longrightarrow h(s)$ admissible

Efficiency: A* explores all states s satisfying the following equation: $\text{PastCost}(s) \leqslant \text{PastCost}(s_{\text{end}}) - h(s)$

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

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_{rel} with costs Cost_{rel}, and satisfies the identity:

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

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

Consistency of relaxed heuristics - Let P_{rel} be a given relaxed problem. By theorem, we have:

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

Tradeoff when choosing heuristic - We have to balance two aspects in choosing a heuristic: (1) Computational efficiency: $h(s) = \text{FutureCost}_{\text{rel}}(s)$ must be easy to compute. It has to produce a closed form, easier search and independent subproblems. (2) Good enough approximation: the heuristic $h(s)$ should be close to $\text{FutureCost}(s)$ 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:

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

Markov Decision Process

Notations Definition - The objective of a Markov decision process is to maximize rewards. It is defined with: (a) a starting state s_{start}, (b) possible actions $\text{Actions}(s)$ from state s, (c) transition probabilities $T(s, a, s')$ from s to s' with action a, (d) rewards $\text{Reward}(s, a, s')$ from s to s' with action a, (e) whether an end state was reached $\text{IsEnd}(s)$, (f) a discount factor $0 \leqslant \gamma \leqslant 1$

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:

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

Policy - A policy π is a function that maps each state s to an action a, i.e. $\pi : s \mapsto a$

Utility - The utility of a path $(s_0, \dots, s_k)$ is the discounted sum of the rewards on that path. In other words,

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

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

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

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:

$$V_\pi(s) = Q_\pi(s, \pi(s))$$

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

Policy evaluation - Given a policy π, policy evaluation is an iterative algorithm that aims at estimating V_π. It is done as follows:

(1) Initialization: for all states s, we have $V_\pi^{(0)}(s) \longleftarrow 0$

(2) Iteration: for t from 1 to T_{PE}, we have

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

with

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

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 $\mathcal{O}(T_{PE} S S')$.

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

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

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

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

Optimal policy - The optimal policy π_{opt} is defined as being the policy that leads to the optimal values. It is defined by:

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

Value iteration - Value iteration is an algorithm that finds the optimal value V_{opt} as well as the optimal policy π_{opt}. It is done as follows:

(1) Initialization: for all states s, we have $V_{\text{opt}}^{(0)}(s) \longleftarrow 0$

(2) Iteration: for t from 1 to T_{VI}, we have

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

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

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.

When unknown transitions and rewards

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

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

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

These estimations will be then used to deduce Q-values, including Q_π and Q_{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:

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

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

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

as well as a stochastic gradient formulation:

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

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:

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

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_{opt}. On each (s, a, r, s', a'), we have:

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

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 π_{act} is computed as follows:

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

Game Playing

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: (1) a starting state s_{start} (2) possible actions $\text{Actions}(s)$ from state s (3) successors $\text{Succ}(s, a)$ from states s with actions a (4) whether an end state was reached $\text{IsEnd}(s)$ (5) the agent's utility $\text{Utility}(s)$ at end state s (6) the player $\text{Player}(s)$ 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: (1) Deterministic policies, noted $\pi_p(s)$, which are actions that player p takes in state s. (2) 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 $V_{\text{em}}(s)$ is the maximum expected utility of any agent policy when playing with respect to a fixed and known opponent policy π_{opp}. It is computed as follows:

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

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:

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

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 π_{agent}, then the agent would be no better off.

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

Property 2: if the opponent changes its policy from π_{min} to π_{opp}, then he will be no better off.

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

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.

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

In the end, we have the following relationship:

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

Speeding Up Minimax

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

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 $\text{Succ}(s, a)$. For each (s, a, r, s'), the update is done as follows:

$$w \longleftarrow w - \eta\big[V(s, w) - (r + \gamma V(s', w))\big]\nabla_w V(s, w)$$

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:

(1) A pure strategy is a single action: $a \in$ Actions

(2) A mixed strategy is a probability distribution over actions: $\forall a \in$ Actions, $\quad 0 \leqslant \pi(a) \leqslant 1$

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:

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

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:

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

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 $(\pi_A^\star, \pi_B^\star)$ such that no player has an incentive to change its strategy. We have: $\forall \pi_A, V_A(\pi_A^\star, \pi_B^\star) \geqslant V_A(\pi_A, \pi_B^\star)$

$$\forall \pi_B, V_B(\pi_A^\star, \pi_B^\star) \geqslant V_B(\pi_A^\star, \pi_B)$$

CSP

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

Factor graphs

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

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

Constraint satisfaction problem - A constraint satisfaction problem (CSP) is a factor graph where all factors are binary; we call them to be constraints: $\forall j \in [\![1, m]\!], \quad f_j(x) \in \{0, 1\}$

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 $\text{Weight}(x) = 1$, i.e. all constraints are satisfied.

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.

Arc consistency - We say that arc consistency of variable X_l with respect to X_k is enforced when for each $x_l \in$ Domain$_l$:

- unary factors of X_l are non-zero,
- there exists at least one $x_k \in$ 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.

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 $K \in \{1, \ldots, b^n\}$ controls the tradeoff between efficiency and accuracy. This algorithm has a time complexity of $O(n \cdot Kb\log(Kb))$.

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 $u \in$ 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.

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: $A \perp\!\!\!\perp B$

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: $A \perp\!\!\!\perp B | C$

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 $f_1, \ldots, f_k$ that depend on X_i
- Remove X_i and $f_1, \ldots, f_k$
- Add $g_j(x)$ for $j \in \{1, \ldots, k\}$ defined as: $g_j(x) = f_j(x \cup \{X_i : v\})$

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 $C =$ MarkovBlanket(A) and $B = X \backslash (A \cup C)$. Then we have:

$$A \perp\!\!\!\perp B | C$$

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 $f_{i,1}, \ldots, f_{i,k}$ that depend on X_i
- Remove X_i and $f_{i,1}, \ldots, f_{i,k}$
- Add $f_{\text{new},i}(x)$ defined as: $f_{\text{new},i}(x) = \max_{x_i} \prod_{l=1}^{k} f_{i,l}(x)$

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

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

Bayesian Networks

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 $X = (X_1, \ldots, X_n)$ as a product of local conditional distributions, one for each node:

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

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

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.

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 a 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 $k \in \{1, \ldots, L\}$ in the case of an HMM of size L. To do so, we proceed in 3 steps:

Step 1 : for $i \in \{1, \ldots, L\}$, 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 $i \in \{L, \ldots, 1\}$, 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 $i \in \{1, \ldots, L\}$, compute $S_i(h_i) = \dfrac{F_i(h_i) B_i(h_i)}{\sum_{h_i} F_i(h_i) B_i(h_i)}$

with the convention $F_0 = B_{L+1} = 1$. From this procedure and these notations, we get that $P(H = h_k | E = e) = S_k(h_k)$

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 $i \in \{1, \ldots, n\}$ until convergence:

- For all $u \in$ 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. $\max_\theta \prod_{x \in \mathcal{D}_{\text{train}}} p(X = x; \theta)$

Laplace smoothing - For each distribution d and partial assignment $(x_{\text{Parents}(i)}, x_i)$, add λ to 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: $q(h) = P(H = h | E = e; \theta)$

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

Basics

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:

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

Set of models - $\mathcal{M}(f)$ denotes the set of models w that satisfy formula f. Mathematically speaking, we define it as follows: $\mathcal{M}(f) = \{w \mid \mathcal{I}(f, w) = 1\}$

Knowledge Base

Definition - The knowledge base KB is the conjunction of all formulas that have been considemagenta 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:

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

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. :

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

Satisfiability - The knowledge base KB is said to be satisfiable if at least one model w satisfies all its constraints. In other words: KB satisfiable $\Longleftrightarrow \mathcal{M}(\text{KB}) \neq \varnothing$

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

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 $f_1, \ldots, f_k$ and conclusion g is written: $\dfrac{f_1, \ldots, f_k}{g}$

Forward inference algorithm - From a set of inference rules Rules, this algorithm goes through all possible $f_1, \ldots, f_k$ 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:

Soundness: $\{f \mid \text{KB} \vdash f\} \subseteq \{f \mid \text{KB} \models f\}$

Completeness: $\{f \mid \text{KB} \vdash f\} \supseteq \{f \mid \text{KB} \models f\}$

Propositional Logic

Horn clause - By noting $p_1, \ldots, p_k$ and q propositional symbols, a Horn clause has the form: $(p_1 \wedge \ldots \wedge p_k) \longrightarrow q$

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

Modus ponens - For propositional symbols $f_1, \ldots, f_k$ and p, the modus ponens rule is written: $\dfrac{f_1, \ldots, f_k, \quad (f_1 \wedge \ldots \wedge f_k) \longrightarrow p}{p}$

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	$\leftrightarrow$	$f \leftrightarrow g$	$(f \to g) \wedge (g \to f)$
	$\to$	$f \to g$	$\neg f \vee g$
	$\neg\neg$	$\neg\neg f$	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 rule - For propositional symbols $f_1, \ldots, f_n$, and $g_1, \ldots, g_m$ as well as p, the resolution rule is written: $\dfrac{f_1 \vee \ldots \vee f_n \vee p, \quad \neg p \vee g_1 \vee \ldots \vee g_m}{f_1 \vee \ldots \vee f_n \vee g_1 \vee \ldots \vee g_m}$

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

First Order Logic

Model - A model w in first-order logic maps:

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

Horn clause - By noting $x_1, \ldots, x_n$ variables and $a_1, \ldots, a_k, b$ atomic formulas, the first-order logic version of a horn clause has the form:
$\forall x_1, \ldots, \forall x_n, \quad (a_1 \wedge \ldots \wedge a_k) \to b$

Substitution - A substitution θ maps variables to terms and Subst$[\theta, f]$ 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: Unify$[f, g] = \theta$ s.t. Subst$[\theta, f] =$ Subst$[\theta, g]$

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

Modus ponens - By noting $x_1, \ldots, x_n$ variables, $a_1, \ldots, a_k$ and $a'_1, \ldots, a'_k$ atomic formulas and by calling $\theta =$ Unify$(a'_1 \wedge \ldots \wedge a'_k, a_1 \wedge \ldots \wedge a_k)$ the first-order logic version of modus ponens can be written: $\dfrac{a'_1, \ldots, a'_k \quad \forall x_1, \ldots, \forall x_n (a_1 \wedge \ldots \wedge a_k) \to b}{\text{Subst}[\theta, b]}$

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

Resolution rule - By noting $f_1, \ldots, f_n, g_1, \ldots, g_m, p, q$ formulas and by calling $\theta =$ Unify(p, q), the first-order logic version of the resolution rule can be written:
$\dfrac{f_1 \vee \ldots \vee f_n \vee p, \quad \neg q \vee g_1 \vee \ldots \vee g_m}{\text{Subst}[\theta, f_1 \vee \ldots \vee f_n \vee g_1 \vee \ldots \vee g_m]}$

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

Farmer Kim

Farmer Kim wants to install a set of sprinklers to water all his crops in the most cost effective manner and has hired you as a consultant. Specifically, he has a rectangular plot of land, which is broken into W × H cells. For each cell (i, j), let $C_{i,j} \in 0, 1$ denote whether there are crops in that cell that need watering. In each cell (i, j), he can either install $(X_{i,j} = 1)$ or not install $(X_{i,j} = 0)$ a sprinkler. Each sprinkler has a range of R, which means that any cell within Manhattan distance of R gets watered. The maintenance cost of the sprinklers is the sum of the Manhattan distances from each sprinkler to his home located at $(1, 1)$. Recall that the Manhattan distance between (a_1, b_1) and (a_2, b_2) is $|a_1 - a_2| + |b_1 - b_2|$. Naturally, Farmer Kim wants the maintenance cost to be as small as possible given that all crops are watered.

Farmer Kim actually took CS221 years ago, and remembered a few things. He says: "I think this should be formulated as a factor graph. The variables should be $X_{i,j} \in 0, 1$ for each cell (i, j). But here's where my memory gets foggy. What should the factors be?" Let $X = X_{i,j}$ denote a full assignment to all variables $X_{i,j}$. Your job is to define two types of factors: $\cdot f_{i,j}$: ensures any crops in (i, j) are watered, $\cdot f_{\text{cost}}$: encodes the maintenance cost, so that a maximum weight assignment corresponds to a valid sprinkler installation with minimum maintenance cost.

Solution: For each cell (i, j), let $f_{i,j}$ encode whether the crops (if they exist) in (i, j) are watered:

$$f_i, j(X) = [C_{i,j} = 0 \text{ or } \min_{i',j' : X_{i',j'} = 1} |i' - i| + |j' - j| \leq R]$$

We define the next factor to the exponentiated negative minimum cost, so that the factor is non-negative and that maximizing the weight corresponds to minimizing the maintenance cost:

$$f_{\text{cost}}(X) = \exp\Big(-\sum_{i',j' : X_{i',j'} = 1} |i' - 1| + |j' - 1|\Big)$$

Rafting

You are going on a rafting trip! The river is modeled as a grid of positions (x, y), where $x \in -m, -(m-1), \ldots, 0, \ldots, (m-1)$, m represents the horizontal offset from the middle of the river and $y \in 0, \ldots, n$ is how far down the river you are. To make things more challenging, there are a number of rocks in the river: For each position (x, y), let $R(x, y) = 1$ if there is a rock and 0 otherwise. You can assume that the start and end positions do not have rocks.

Here's how you can control the raft. From any position (x, y), you can:

- go straight down to $(x, y + 1)$ (which takes 1 second),
- veer left to $(x - 1, y + 1)$ (which takes 2 seconds), or
- veer right to $(x + 1, y + 1)$ (which takes 2 seconds).

If the raft enters a position with a rock, there is an extra 5 second delay. The raft starts in $(0, 0)$ and you want to end up in any position (x, y) where $y = n$.

DP Solution: At each point in time, you can take an action a ⊠ −1, 0, +1, which has cost $|a| + 1$ (just a succinct way of representing the cost). In addition, you incur an extra $5R(x, y)$ for hitting a rock. The full recurrence is:

$$C(x, y) = 5R(x, y) +$$

$$\begin{cases} 0 & \text{if } y = n, \\ \min_{a \in -1, 0, +1} (|a| + 1 + C((x + a, y + 1))) & \text{otherwise.} \end{cases}$$

MDP: First, let us figure out what the state should be. At position (x, y), you have knowledge about each $R(x', y')$ for all x', y' for which $y' \leq y + 1$ and for all x'. This is a lot of information to remember (exponential in mn). The key is that the only relevant bits of information are $R(x', y')$ for the positions that you might move to, of which there are 3. Let $s = (x, y, r_{-1}, r_0, r_1)$, where r_a denotes whether position $(x + a, y + 1)$ has a rock or not. Thus, there are only $2^3 mn$ possible states that you can be in at any given time.

- $s_{\text{start}} = (0, 0, 0, 0, 0)$
- Actions(s) = $\{a \in -1, 0, +1$: a keeps you on the grid$\}$
- T(s, a, s') $= \alpha^k (1 - \alpha)^{3-k}$ if $x' = x + a, y' = y + 1$ and 0 otherwise, where we define $k = r'_{-1} + r'_0 + r'_1$ to be the number of rocks that will appear next.
- Reward(s, a, s') = $-[(|a| + 1) + 5r_a]$, where the first term is the cost of taking the action and the second term is the cost (0 or 1) of hitting a rock.
- IsEnd(s) = $[y = n]$
- $\gamma = 1$

As before, assume that you don't have the map describing the position of the rocks, but you know that the probability distribution over the map is $R(x, y) = 1$ independently with probability α.

Consider the following two scenarios:

- A genie reveals the entire map to you right as you get on your raft. Let T_1 be this expected minimum time of getting to an end goal.
- There is sadly no genie, and you have to use your own eyes to look at the position of the next row as you're rafting. Let T2 be this minimum expected time of getting to an end goal.

Prove that $T1 \leq T2$. (Intuitively this should be true; you must argue it mathematically.)

Proof Solution First, note that $\sum_a p(a) \min_b F(a, b) \leq \min_b p(a) F(a, b)$, because in the first case you get a different b for each a and in the second, you don't. Let $R_1, \ldots, R_n$ be the rows of R, and let $a_1, \ldots, a_n$ be the n actions that you take to get from row 0 to row n.

Let $C(R_1, \ldots, R_n, a_1, \ldots, a_n)$ be the cost of your journey, and let $p(R_y)$ be the probability of having a particular configuration of rocks in row y. In the first case, we are computing an expectation over a minimum:

$$\sum_{R_1, \ldots, R_n} p(R_1) \ldots p(R_n) \min_{a_1, .., a_n} C(R_1, .., R_n, a_1, .., a_n).$$

In the second case, we are interleaving the minimum with the expectation:

$$\sum_{R_1} p(R_1) \min_{a_1} .. \sum_{R_n} p(R_n) \min_{a_n} C(R_1, .., R_n, a_1, .., a_n).$$

Having all the mins on the inside can only make the expected time smaller. In other words, going second in a game is always preferable.

Suppose we don't actually know how long it takes to go over rocks or what the distribution of rocks is, so we will use Q-learning to learn a good policy.

- Suppose the state s includes the position (x, y) and the map that's revealed so far, i.e. $R(x', y')$ for all x' and $y' \leq y + 1$.
- The actions are $a \in -1, 0, +1$, corresponding to going left, straight, or right.
- The reward is the negative time it takes to travel from state s to the new state s'.
- Assume the discount $\gamma = 1$.

For each state s and action a, let $H(s, a) = 1$ if a causes the raft to hit a rock and 0 otherwise.
Now define the approximate Q-function to be:
$Q(s, a; \alpha, \beta) = \alpha H(s, a) + \beta$,
where α and β are parameters to be learned. Suppose we sample once from an exploration policy, which led to the trajectory shown in Figure 1.

(i) Write down the Q-learning updates on α on experience (s, a, r, s') using a step size η.
Solution For updating α:
α⊠$\alpha - \eta[(\alpha H(s, a) + \beta) - (r + \gamma \max_{a'} (\alpha H(s', a') + \beta))] H(s, a)$
$= \alpha - \eta[\alpha H(s, a) - (r + max_{a'} \alpha H(s', a'))] H(s, a)$.
For updating β:
β⊠$\beta - \eta[(\alpha H(s, a) + \beta) - (r + \gamma max_{a'} (\alpha H(s', a') + \beta))]$
$= \beta - \eta[\alpha H(s, a) - (r + max_{a'} \alpha H(s', a'))]$.

(ii) On how many of the $n = 10$ updates could α change its value?
Updates Solution The paramater α is updated only when $H(s, a) = 1$, which means that we hit a rock. This happens twice.

- Each other car's policy chooses the action a that minimizes its immediate cost (not your cost) plus the distance from xi + a to (W, H). Any ties are broken randomly.

Solution The greedy policy is a known stochastic policy. Due to the randomness, it is not a search problem, but can be cast as an MDP. The default algorithm for computing optimal policies is value iteration. But because the MDP is acyclic, we could also compute this using a recursive dynamic programming.

- Each other car's policy uses the learned policy produced from part (d).

Solution This is again a known stochastic policy (the fact that it was learned is irrelevant), so the answer is the same as for the greedy policy. · Each other car's policy is optimally minimizing your cost (which might be the case if you had a siren on your car).

Solution All cars are now trying to minimize your cost, so this is a search problem. Which can be solved using UCS or A* (all costs are non-negative), or dynamic programming (since the state graph is acyclic).

- Each other car's policy is optimally maximizing your cost.

Solution This is a classic turn-based zero-sum game (very similar to Pac-Man with cars instead of ghosts). We would compute the recurrence using dynamic programming.

- Each other car's policy is trying to minimize its own cost.

Solution This is a turn-based non-zero-sum game. These games in general don't have optimal policies, but merely Nash equilibria. How to compute them is outside the scope of this class.

- To decrease training error, would you want more or less data?

Solution Less. Fewer data points are easier to fit.

- To decrease training error, would you want to add or remove features?

Solution Add. More features makes it easier to fit the data.

- To decrease training error, would you want to make the set of hypotheses smaller or larger?

Solution Larger. More hypotheses makes it easier to fit the data.

Define the linear predictor (parametrized by numbers w, b) to be $f(x) = sign(wx + b)$ with the associated zero-one loss and hinge loss, respectively:
$Loss_{0-1}(x, y, w, b) = 1[f(x) \neq y]$,
$Loss_{\text{hinge}}(x, y, w, b) = \max(0, 1 - y(wx + b))$.
Define the total training zero-one and hinge losses as the following:

$$\text{TrainLoss}_{0-1}(w, b) = \sum_{(x,y) \in D_{\text{train}}} Loss_{0-1}(x, y, w, b),$$

$$\text{TrainLoss}_{\text{hinge}}(w, b) = \sum_{(x,y) \in D_{\text{train}}} Loss_{\text{hinge}}(x, y, w, b).$$

Solution The derivative of the max is an indicator function and then we use the chain rule and calculate the derivative of the argument.

$$\partial \text{TL}_{\text{hinge}}(w, b) / \partial w = \sum_{(x,y) \in D_{\text{train}}} 1[1 - y(wx + b) > 0](-yx)$$

$$\partial \text{TL}_{\text{hinge}}(w, b) / \partial b = \sum_{(x,y) \in D_{\text{train}}} 1[1 - y(wx + b) > 0](-y)$$