ECOLE POLYTECHNIQUE
ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2025

MARDI 15 AVRIL 2025
14h00 - 18h00

FILIERES MP-MPI - Epreuve n° 4

INFORMATIQUE A (XULSR)

Durée : 4 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Cette composition ne concerne qu'une partie des candidats de la filière MP, les autres candidats effectuant simultanément la composition de Physique et Sciences de l'Ingénieur.
Pour la filière MP, il y a donc deux enveloppes de Sujets pour cette séance.

Arbres de classification d'arbres

Le sujet comporte 12 pages.

Vue d'ensemble du sujet.

Ce sujet porte sur la réalisation d'arbres d'identification des plantes basés sur des observations morphologiques. À partir d'une base de connaissance botaniste, il s'agit de réaliser un arbre dont les feuilles sont des diagnostics (typiquement des noms d'espèce) et les nœuds des *caractères morphologiques* (par exemple la couleur des fleurs). Un nœud a autant de fils que le caractère a de valeurs possibles (jaune, bleue, etc.).

Étant donnée une base de connaissance, il s'agit de construire un arbre dont la hauteur moyenne est la plus petite possible, ce qui revient à minimiser le nombre moyen d'observations nécessaires pour identifier une plante. La difficulté revient donc à choisir à chaque étape le caractère le plus *discriminant*, c'est-à-dire celui qui va le mieux séparer l'espace de recherche.

Pour tenir compte de la diversité au sein d'une espèce, nous adoptons une approche probabiliste basée sur un raisonnement bayésien : la valeur d'un caractère morphologique pour une espèce donnée sera probabiliste. Par exemple, la couleur des fleurs d'une certaine espèce pourra être bleue ou jaune, avec certaines probabilités.

La partie I introduit les éléments de raisonnement bayésien nécessaires à la construction des arbres. La partie II propose une approche par énumération pour générer un arbre qui minimise la hauteur moyenne. La partie III propose une heuristique basée sur l'entropie de Shannon pour éviter la construction de tous les arbres. La partie IV revient sur le calcul exact en proposant une optimisation permettant de réduire l'espace de recherche. Les parties III et IV sont indépendantes l'une de l'autre. Enfin, la partie V propose une approche logique pour étendre la notion de caractère. Cette dernière partie ne dépend que de la première.

Rappels de probabilités

Une **mesure** sur un ensemble fini X est une fonction $\varphi : X \to \mathbb{R}^+$. Sa **masse** est la quantité $\sum_{x \in X} \varphi(x)$. Une mesure de masse 1 est appelée une **distribution de probabilité**, ou simplement **distribution**. On note $D(X)$ l'ensemble des distributions sur l'ensemble fini X. Pour $x \in X$, la **dirac** en x, notée δ_x, est la distribution qui associe à x le poids 1, et zéro à tous les autres éléments. Étant donnés une famille de distributions $d_1, ..., d_n \in D(X)$ et des poids $p_1, ..., p_n \in [0; 1]$ tels que $\sum_{1 \leq i \leq n} p_i = 1$, on note $p_1 \cdot d_1 + \ldots + p_n \cdot d_n$ la distribution $d \in D(X)$ telle que $d(x) = p_1 \times d_1(x) + \cdots + p_n \times d_n(x)$ pour tout $x \in X$.

Si X est un ensemble, on notera $\mathscr{P}(X)$ l'ensemble de ses parties.

Rappels d'OCaml

Dans le reste du sujet, on pourra utiliser les fonctions suivantes de la bibliothèque standard OCaml pour les listes :

- `List.hd [e1;...;en]` renvoie `e1`, et `List.hd []` lève une exception.
- `List.tl [e1;...;en]` renvoie `[e2;...;en]`, et `List.tl []` lève une exception.
- `List.length l` renvoie la longueur (nombre d'éléments) de la liste `l`.
- `List.init n f` renvoie la liste `[f 0;...;f (n-1)]` pour `n` positif.
- `List.assoc k [(k1,v1);...;(kn,vn)]` renvoie le premier `vi` tel que `k = ki` s'il existe, et lève l'exception `Not_found` sinon.
- `List.find f [e1;...;en]` renvoie le premier `ei` tel que `f ei = true` s'il existe, et lève l'exception `Not_found` sinon.
- `List.map f [e1;...;en]` renvoie `[f e1;...;f en]`.
- `List.filter f l` renvoie la liste des éléments `e` de `l` tels que `f e = true`, dans l'ordre.
- `List.concat [l1;...;ln]` renvoie `l1@...@ln`, la concaténation des listes `li`.
- `List.concat_map f l` renvoie `List.concat (List.map f l)`.
- `List.fold_left f a0 [e1;...;en]` renvoie `an` où `ak+1 = f ak ek+1` pour $0 \leq k < n$.

Pour les tableaux, on pourra utiliser les fonctions suivantes :

- `Array.length a` renvoie la longueur (nombre d'éléments) du tableau `a`.
- `Array.init n f` renvoie le tableau `[|f 0;...;f (n-1)|]` pour `n` positif.
- `Array.of_list [e1;...;en]` renvoie le tableau `[|e1;...;en|]`.

On rappelle enfin que la fonction `log : float -> float` est disponible en OCaml pour calculer un logarithme naturel.

Partie I : Description probabiliste des plantes

On se donne un nombre de caractères $N \in \mathbb{N}^*$. Un **caractère** sera un entier $i \in \{0, 1, \ldots, N-1\}$. Pour chaque caractère i, on se donne un ensemble fini V_i de **valeurs** possibles pour ce caractère.

Pour nos exemples, nous prendrons $N = 2$ et deux valeurs pour chaque caractère : $V_0 = \{4, 5\}$ représentera le nombre de pétales des fleurs, et $V_1 = \{\mathsf{bleu}, \mathsf{blanc}\}$ représentera leur couleur.

Nous utiliserons une représentation probabiliste des espèces pour modéliser la variabilité intra-espèce, ainsi que la variabilité des interprétations lors des observations : on peut trouver au sein d'une même espèce des individus à fleurs bleues et d'autres à fleurs blanches ; par ailleurs, une couleur bleue très claire pourra être interprétée comme bleue ou blanche en fonction de l'observateur. Ainsi, chaque espèce sera décrite par des valeurs possibles pour chaque caractère, organisées en distributions de probabilité : une **espèce** est un N-uplet $s = (s_0, ..., s_{N-1})$ où $s_i \in D(V_i)$. Intuitivement, $s_i(v)$ représente la probabilité que la valeur v soit observée pour le caractère i chez un individu de l'espèce. On suppose donné un ensemble fini d'espèces $\mathscr{S}$.

Dans nos exemples, $\mathscr{S}$ sera composé des trois espèces suivantes, décrites en utilisant des diracs δ :

$$\begin{aligned} \mathsf{myosotis} &= (\delta_5,\ 0,8 \cdot \delta_{\mathsf{bleu}} + 0,2 \cdot \delta_{\mathsf{blanc}}) \\ \mathsf{lin} &= (\delta_5,\ \delta_{\mathsf{bleu}}) \\ \mathsf{gaillet} &= (\delta_4,\ \delta_{\mathsf{blanc}}) \end{aligned}$$

On définit l'ensemble Ω suivant, dont les éléments modélisent des individus, décrits par leur espèce et les valeurs de leurs caractères :

$$\Omega \stackrel{\text{def}}{=} \{(s, v_0, \ldots, v_{N-1}) \mid s \in \mathscr{S}, v_i \in V_i \text{ pour tout } i\}$$

On considèrera l'espace probabilisable $(\Omega, \mathscr{P}(\Omega))$ obtenu en munissant Ω de la tribu discrète. On utilisera plusieurs lois de probabilité sur cet espace, définies en fonction d'un **état** $\sigma \in D(\mathscr{S})$: intuitivement, l'état indiquera la probabilité d'observer chaque espèce. La **loi de probabilité** P^σ **induite par un état** σ est définie comme l'unique loi satisfaisant l'équation suivante pour tout $(s, v_0, \ldots, v_{N-1}) \in \Omega$:

$$P^\sigma(\{(s, v_0, \ldots, v_{N-1})\}) \ = \ \sigma(s) \times \prod_{0 \le i < N-1} s_i(v_i)$$

On notera S la variable aléatoire qui à $(s, v_0, \ldots, v_{N-1}) \in \Omega$ associe s. Pour chaque caractère i, on notera C_i la variable aléatoire qui à $(s, v_0, \ldots, v_{N-1}) \in \Omega$ associe v_i. On pourra ainsi écrire, par exemple, $P^\sigma(C_i = v \mid S = s)$, qui représente (dans l'état σ) la probabilité (conditionnelle) que le caractère i prenne la valeur v chez un individu de l'espèce s.

Préliminaires probabilistes. On établit tout d'abord quelques résultats élémentaires sur lesquels s'appuiera notre méthode de classification.

Question 1. On considère l'état $\sigma_0 = \{\mathsf{gaillet} : 50\%, \mathsf{myosotis} : 30\%, \mathsf{lin} : 20\%\}$ sur les espèces de notre exemple. Quelle est la probabilité $P^{\sigma_0}(C_1 = \mathsf{blanc})$ d'observer des fleurs blanches dans cet état ? Pour chaque espèce s, que vaut la probabilité $P^{\sigma_0}(S = s \mid C_1 = \mathsf{blanc})$ d'observer un individu de l'espèce s sachant que celui-ci a des fleurs blanches ? Aucune justification n'est attendue.

Question 2. Étant donnés un état $\sigma \in D(\mathscr{S})$, une espèce $s \in \mathscr{S}$, un caractère i et une valeur $v \in V_i$, exprimer simplement les probabilités $P^\sigma(S = s)$ et $P^\sigma(C_i = v)$, ainsi que $P^\sigma(C_i = v \mid S = s)$ quand elle est bien définie. Les réponses devront être justifiées.

Question 3. Étant donnés un état $\sigma \in D(\mathscr{S})$, un caractère i, une valeur $v \in V_i$ et une espèce $s \in \mathscr{S}$, exprimer $P^\sigma(S = s \mid C_i = v)$. Sous quelle condition cette probabilité est-elle bien définie ?

Pour un état σ, un caractère i et une valeur $v \in V_i$, on définit **l'état mis à jour** $\sigma[i := v]$ comme la distribution qui à chaque espèce s associe $P^\sigma(S = s \mid C_i = v)$.

Question 4. Pour un état σ, deux caractères $i \neq j$ et des valeur $v \in V_i$ et $v' \in V_j$, montrer que $P^\sigma(S = s \mid C_i = v, C_j = v') = P^{\sigma[i:=v]}(S = s \mid C_j = v')$ quand ces probabilités sont bien définies.

Le résultat précédent nous indique que, si l'on souhaite évaluer dans un état σ la probabilité d'observer une certaine espèce sachant que $C_i = v$ et $C_j = v'$, il nous suffit de calculer une probabilité conditionnée seulement par $C_j = v'$ mais dans l'état $\sigma[i := v]$. En allant plus loin, si l'on pose $\sigma' = \sigma[i := v]$ et $\sigma'' = \sigma'[j := v']$, on a $P^\sigma(S = s \mid C_i = v, C_j = v') = \sigma''(s)$. On admettra que ce résultat se généralise à un nombre arbitraire d'observations. Ainsi, il est possible de procéder à la reconnaissance de plantes en calculant des états mis à jours par des observations successives, jusqu'à atteindre un état de la forme δ_s ou ne plus pouvoir faire de nouvelle observation.

Choix des représentations pour le code. Un caractère sera naturellement représenté par un entier OCaml. On représentera aussi les valeurs par des entiers, en supposant que chaque V_i est de la forme $\{0, 1, \ldots, k_i - 1\}$, ce qui revient à numéroter les valeurs possibles. On pose ainsi :

```
type caractere = int
type valeur = int
```

On suppose donnés le nombre de valeurs pour chaque caractère sous la forme d'un tableau :

```
val num : int array    (* num.(i) = k_i, i.e., V_i = {0,1,...,num.(i)-1} *)
```

Enfin, on considèrera que la liste des caractères `[0;1;...;N-1]` est définie en variable globale à partir du tableau `num` :

```
let caracteres : caractere list = List.init (Array.length num) (fun i -> i)
```

Pour notre exemple avec deux caractères, on aura ainsi :

```
(* Nombre de valeurs possibles pour le nombre de pétales et pour leur couleur *)
let num = [|2; 2|]

(* Nombre de pétales *)
let nb_petales : caractere = 0
let quatre, cinq = 0, 1

(* Couleur *)
let couleur_petales : caractere = 1
let bleu, blanc = 0, 1
```

Une mesure μ sur X sera représentée par une liste de couples $(x, \mu(x))$ avec $x \in X$ et $\mu(x) > 0$, sans doublons. **En particulier, les éléments de poids nuls ne sont pas inclus dans la liste, et l'ordre n'a pas d'importance.**

```
type 'x mesure = ('x * float) list
let dirac v = [(v, 1.)]
```

Enfin, on codera comme suit les espèces et états :

```
type espece = { name: string; mesures: valeur mesure array }
type etat = espece mesure
```

Pour notre exemple, on aura ainsi :

```
let myosotis : espece =
  { name = "myosotis";
    mesures = [| dirac cinq; [(bleu, 0.8);(blanc, 0.2)] |] }
let lin : espece =
  { name = "lin";
    mesures = [| dirac cinq; dirac bleu |] }
let gaillet : espece =
  { name = "gaillet";
    mesures = [| dirac quatre; dirac blanc |] }
let sigma0 : etat = [(gaillet, 0.5);(myosotis, 0.3);(lin, 0.2)]
```

Question 5. Écrire les fonctions suivantes :

```
val proba_de       : 'x -> 'x mesure -> float
val somme_ponderee : 'x mesure -> ('x -> float) -> float
val repondere      : 'x mesure -> ('x -> float) -> 'x mesure
```

La première renvoie le poids d'un élément selon une mesure donnée. La seconde renvoie la somme d'une mesure μ pondérée par une fonction $f : X \to \mathbb{R}$, définie par $\sum_{x \in X} \mu(x) f(x)$. La troisième repondère une mesure sur X par une fonction $f : X \to \mathbb{R}$: le poids de $x \in X$ passe de $\mu(x)$ à $\mu(x)f(x)$.

Question 6. Écrire deux fonctions

```
val proba_espece : espece -> caractere -> valeur -> float
val proba_etat   : etat -> caractere -> valeur -> float
```

où `proba_espece` s i v renvoie la probabilité $P^{\delta_s}(C_i = v)$ et `proba_etat` σ i v renvoie $P^{\sigma}(C_i = v)$.

Question 7. Écrire une fonction

```
val reponse : etat -> caractere -> valeur -> etat
```

qui, étant donnés un état σ, un caractère i et une valeur $v \in V_i$, renvoie l'état $\sigma[i := v]$ si celui-ci est bien défini, et lève une exception sinon.

Partie II : Arbre optimal

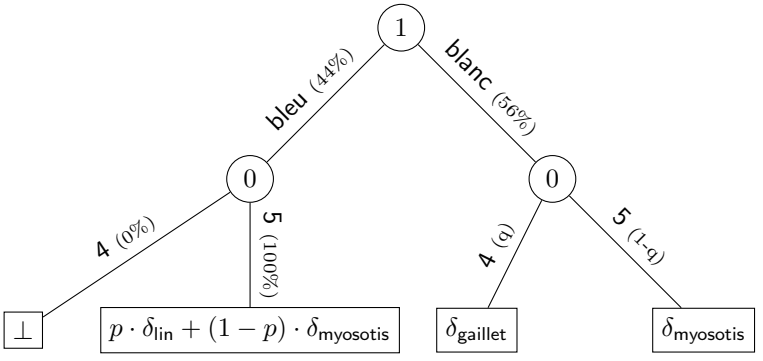
Dans cette seconde partie, on définit la notion d'arbre de décision, et on s'intéresse au calcul d'un arbre de décision optimal en un certain sens. On introduit les types de données suivants pour coder nos arbres :

```
type noeud = {caractere: caractere; enfants: (float * arbre) array}
and arbre =
  | Noeud of noeud       (* On pose une question sur un caractère. *)
  | Feuille of etat      (* On a atteint une feuille, on donne l'état. *)
  | Impossible           (* On a atteint une feuille impossible :
                            aucune espèce ne correspond aux réponses données. *)
```

Les valeurs de type `arbre` ne représentent pas toutes des arbres de décision. Une valeur t de type `arbre` est **un arbre de décision** pour un ensemble de caractères C et un état σ lorsque :

- L'arbre t n'est pas `Impossible`.
- Si t est de la forme `Feuille` σ', alors $\sigma' = \sigma$ et (au moins) l'une des deux conditions suivantes est satisfaite :
  - σ est une distribution dirac (il n'y a plus qu'une seule espèce possible) ;
  - $C = \emptyset$ (il n'y a plus de caractère disponible pour discriminer).
- Si t est de la forme `Noeud {caractere=i;enfants=e}` alors `i` est un élément de C, `e` est un tableau à `num.(i)` éléments, et pour chaque valeur possible $0 \leq \texttt{v} < \texttt{num.(i)}$, `e.(v)` est de la forme $(P^\sigma(C_\texttt{i} = \texttt{v}), t')$ avec :
  - si $P^\sigma(C_\texttt{i} = \texttt{v}) = 0$, alors t' est `Impossible` ;
  - sinon, t' est un arbre de décision pour l'ensemble de caractères $C \setminus \{\texttt{i}\}$ et l'état $\sigma[\texttt{i} := \texttt{v}]$.

Dans le contexte de l'exemple de la partie I, on représente graphiquement ci-dessous un arbre de décision pour l'ensemble complet de caractères $\{0, 1\}$ et l'état σ_0 de la question 1 (pour des valeurs de p et q non précisées) :



Les nœuds sont étiquetés par des caractères et les feuilles par des états ; la notation $\perp$ est utilisée pour représenter les cas impossibles où aucune espèce ne peut correspondre aux valeurs observées. Chaque arc est étiqueté par la valeur du caractère correspondant à l'enfant ainsi que la probabilité que cette valeur soit observée.

On définit la **hauteur moyenne** d'un arbre t, notée $h(t)$, comme suit :

- Si t est de la forme `Feuille` σ ou `Impossible`, alors $h(t) = 0$.

— Si t est de la forme `Noeud {caractere=i;enfants=e}`, et si l'on note $(p_v, t_v) =$ `e.(v)`, alors :

$$h(t) \;=\; 1 + \sum_{0 \leq v < \texttt{num.(i)}} p_v \times h(t_v)$$

La hauteur moyenne correspond au nombre moyen de questions avant d'atteindre une feuille. Sur l'exemple précédent, la hauteur moyenne est de deux.

Question 8. Sur l'exemple de la question 1, énumérer tous les arbres de décision possibles pour les caractères $\{0, 1\}$ et l'état σ_0, et donner leur hauteur moyenne — pour l'arbre déjà représenté ci-dessus, on pourra se contenter de préciser les valeurs de p et q. Quel est l'arbre de décision avec la hauteur moyenne minimale ?

Dans la suite de cette partie, on se propose de calculer naïvement un arbre de décision de hauteur moyenne minimale. On dira qu'un arbre t est **optimal pour** C **et** σ quand c'est un arbre de décision pour C et σ tel que $h(t)$ est minimale parmi les hauteurs moyennes de tous les arbres de décision pour C et σ. Dans un contexte où C et σ sont clairs, on dira simplement que t est optimal.

Question 9. Écrire une fonction

```
val choix_possibles : 'x list array -> 'x array list
```

qui renvoie la liste des tableaux obtenus en choisissant un élément dans chaque liste du tableau d'entrée. Par exemple :

```
choix_possibles [| [1; 2]; [3; 4; 5] |] = [
  [| 1; 3 |]; [| 1; 4 |]; [| 1; 5 |];
  [| 2; 3 |]; [| 2; 4 |]; [| 2; 5 |];
]
```

Question 10. Écrire une fonction qui, étant donné un état initial σ_0, renvoie la liste de tous les arbres de décision possibles pour σ_0 et l'ensemble de tous les caractères :

```
val enumere : etat -> arbre list
```

On rappelle que l'ensemble $\{0, 1, \ldots, N-1\}$ de tous les caractères est représenté par la liste `caracteres` déclarée en variable globale.

Question 11. Écrire une fonction

```
val argmin : 'x list -> ('x -> float) -> 'x
```

telle que `argmin l f` renvoie un élément de `l` qui minimise `f`, si la liste est non-vide. En déduire une fonction qui, étant donné un état initial σ_0, renvoie un arbre de décision optimal pour σ_0 et l'ensemble de tous les caractères :

```
val arbre_optimal : etat -> arbre
```

Partie III : Algorithme glouton

Afin d'éviter l'énumération de tous les arbres de décision dans l'algorithme de la partie précédente, on cherche un algorithme glouton : l'idée générale est de décider localement du caractère à choisir en fonction de l'état courant, plutôt que d'envisager tous les choix possibles. Pour cela, on décide de prendre le caractère qui minimise l'**entropie moyenne**. Avant de définir cette quantité, on introduit l'entropie simple $H(\mu)$ d'une distribution $\mu \in D(X)$, qui est donnée par :

$$H(\mu) \stackrel{\text{def}}{=} -\Big(\sum_{x \in X, \mu(x) \neq 0} \mu(x) \ln(\mu(x))\Big)$$

Question 12. Soit X un ensemble de cardinal $k \in \mathbb{N}$. Donner (en justifiant) les valeurs minimale et maximale de H sur $D(X)$ en fonction de k, ainsi que des distributions les atteignant.

Étant donnés un caractère i et un état σ, on définit l'entropie moyenne de i dans l'état σ, notée $H_i(\sigma)$, comme l'espérance des entropies sur les distributions obtenues en faisant une observation sur le caractère i :

$$H_i(\sigma) \stackrel{\text{def}}{=} \sum_{v \in V_i} P^\sigma(C_i = v) H(\sigma[i := v])$$

Question 13. Écrire une fonction qui renvoie l'entropie moyenne d'un caractère dans un état donné :

```
val score_caractere : etat -> caractere -> float
```

Question 14. Écrire une fonction qui calcule un arbre de décision en suivant l'algorithme glouton qui construit l'arbre en choisissant à la racine le caractère minimisant l'entropie moyenne, et procède récursivement de la même façon pour les sous-arbres :

```
val glouton : etat -> arbre
```

On se propose maintenant de montrer que l'algorithme glouton n'est pas optimal. On considère une situation avec deux caractères, $V_0 = V_1 = \{\mathsf{oui}, \mathsf{non}\}$, et trois espèces a, b et c :

$$\begin{array}{rcl} a & = & (\delta_{\mathsf{oui}},\ 0,5 \cdot \delta_{\mathsf{oui}} + 0,5 \cdot \delta_{\mathsf{non}}) \\ b & = & (\delta_{\mathsf{non}},\ \delta_{\mathsf{oui}}) \\ c & = & (\delta_{\mathsf{non}},\ \delta_{\mathsf{non}}) \end{array}$$

On considère enfin l'état initial σ_0 suivant :

$$a : 20\%,\ b : 40\%,\ c : 40\%$$

L'espèce a est donc plus rare que les espèces b et c qui sont plus communes.

Question 15. Dessiner les arbres de décision possibles pour l'état σ_0 et les caractères $\{0, 1\}$, et déterminer l'arbre de décision optimal.

Question 16. Calculer les entropies moyennes $H_0(\sigma_0)$ et $H_1(\sigma_0)$. Que peut-on en conclure ?
On admettra que $\ln(5) < \frac{12}{5} \ln(2)$.

Partie IV : Optimisation de l'algorithme naïf

On cherche à optimiser l'algorithme naïf de la partie II en évitant certains calculs inutiles. Pour permettre cela, on commence par changer la méthode de recherche d'un arbre optimal, en évitant de construire la liste de tous les arbres de décision possibles. La nouvelle méthode s'appuiera sur la fonction `arbre_optimal_avec_oracle` donnée en Figure 1. Le but de cette fonction est, étant donnés un état σ et une liste de caractères C, de renvoyer $(t, h(t))$ où t est un arbre optimal pour C et σ. Il faut bien noter que cette fonction n'est pas récursive, mais qu'elle dispose d'un argument supplémentaire `oracle` dont les appels pourront correspondre à des appels récursifs : il s'agit de *récursion ouverte*.

Question 17. Donner des hypothèses (pré-conditions) aussi précises que possible sur les arguments C, σ et `oracle` de la fonction `arbre_optimal_avec_oracle`, sous lesquelles la fonction termine toujours et renvoie bien $(t, h(t))$ où t est un arbre optimal pour C et σ. On veillera à répondre à cette question après avoir réfléchi aux deux suivantes, où il faudra démontrer puis exploiter la correction de la fonction par rapport à la spécification énoncée ici.

Question 18. Démontrer que la fonction satisfait bien cette spécification, en veillant notamment à énoncer clairement l'invariant de boucle.

Question 19. Écrire une fonction, récursive cette fois-ci, et basée sur `arbre_optimal_avec_oracle`, qui renvoie un arbre de décision optimal pour un état et une liste de caractères donnés :

```
val arbre_optimal : etat -> caractere list -> arbre * float
```

Donner sa spécification et montrer qu'elle la satisfait.

Question 20. On cherche maintenant à optimiser notre algorithme en interrompant la génération d'un arbre dès que son score dépasse celui de l'`arbre_optimal` courant. Pour cela, écrire une nouvelle fonction

```
val arbre_optimal_opt : etat -> caractere list -> float -> (arbre * float) option
```

qui prend un paramètre supplémentaire représentant le score à ne pas dépasser, et renvoie `None` si tous les arbres de décision possibles sont de hauteur moyenne supérieure au score maximal, ou bien un arbre optimal de hauteur moyenne inférieure s'il en existe un. La preuve de correction n'est pas demandée.

```
let arbre_optimal_avec_oracle
      (oracle : etat -> caractere list -> arbre * float)
      (etat : etat)
      (caracteres : caractere list) : arbre * float =
  if caracteres = [] then
    (Feuille etat, 0.)
  else
    let caracteres_a_tester = ref caracteres in
    let score_optimal = ref infinity in
    let arbre_optimal = ref None in
    while !caracteres_a_tester <> [] do
      let caractere = List.hd !caracteres_a_tester in
      let score_total = ref 1. in
      let enfants : (float * arbre) array =
        Array.init num.(caractere) (fun valeur ->
          let p_valeur = proba_etat etat caractere valeur in
          let sous_arbre, score_sous_arbre =
            if p_valeur = 0. then Impossible,0. else
              let etat' = reponse etat caractere valeur in
              let caracteres_restants =
                List.filter (fun c -> c <> caractere) caracteres
              in
              oracle etat' caracteres_restants
         in
         score_total := !score_total +. p_valeur *. score_sous_arbre;
         (p_valeur, sous_arbre))
      in
      if !score_total < !score_optimal then
        (arbre_optimal := Some (Noeud {caractere=caractere;enfants=enfants});
         score_optimal := !score_total);
      caracteres_a_tester := List.tl !caracteres_a_tester
    done;
    match !arbre_optimal with
    | Some a -> a, !score_optimal
    | None -> assert false
```

FIGURE 1 – Fonction `arbre_optimal_avec_oracle`.

Partie V : Observations complexes et formules logiques

Pour aller plus loin, on se propose de représenter des observations complexes sous forme de formules logiques construites à partir des caractères, par exemple "la fleur est bleue et a cinq pétales". La syntaxe de cette logique est la suivante :

$$\varphi ::= c_i \in V \mid \varphi_1 \wedge \ldots \wedge \varphi_m \mid \varphi_1 \vee \ldots \vee \varphi_m \qquad (0 \leq i < N,\ V \subseteq V_i,\ m \geq 0)$$

Cela signifie que l'ensemble des formules est le plus petit ensemble contenant les formules **atomiques** de la forme $c_i \in V$ où $0 \leq i < N$ et $V \subseteq V_i$, et que si $\varphi_1, ..., \varphi_m$ sont des formules pour $m \geq 0$, alors la **conjonction** $\varphi_1 \wedge \cdots \wedge \varphi_m$ et la **disjonction** $\varphi_1 \vee \cdots \vee \varphi_m$ sont aussi des formules. Une formule $c_i \in V$ signifie intuitivement que le caractère i prend une valeur dans l'ensemble $V \subseteq V_i$. On définit la formule $\bot$ (la formule fausse) comme la disjonction vide, et $\top$ (la formule vraie) comme la conjonction vide.

Par exemple, avec les ensembles V_0 et V_1 de l'exemple utilisé en partie I, $c_1 \in \{\mathsf{bleu}\} \wedge c_0 \in \{5\}$ est une formule, correspondant à l'énoncé "la fleur est bleue et a cinq pétales". Ou encore, si $V_i = \{0, 1, 2, 3, 4\}$, $c_i \in \{1\} \wedge c_i \in \{2, 3\}$ est une formule, correspondant à l'énoncé contradictoire "le caractère i vaut 1, et il vaut 2 ou 3".

On définit la taille d'une formule φ, notée $|\varphi|$, comme suit :

$$\begin{array}{lcl} |c_i \in V| & = & 1 + |V| \\ |\varphi_1 \wedge \ldots \wedge \varphi_m| & = & 1 + m + \sum_{1 \leq i \leq m} |\varphi_i| \\ |\varphi_1 \vee \ldots \vee \varphi_m| & = & 1 + m + \sum_{1 \leq i \leq m} |\varphi_i| \end{array}$$

Dans un premier temps, nous définissons quand une situation certaine (dite déterministe) satisfait une certaine formule. On appelle **modèles déterministes** les éléments de $V_0 \times ... \times V_{N-1}$. On notera $\vec{v} \in \vec{V}$ pour signifier que $\vec{v}$ est un tel modèle, et dans ce cas on notera v_i la i-ème composante de $\vec{v}$, ce qui revient à dire que $\vec{v} = (v_0, \ldots, v_{N-1})$. On définit une relation de **satisfaction** entre les modèles déterministes et les formules, notée $\vec{v} \models \varphi$, comme suit :

$$\begin{array}{lll} \vec{v} \models c_i \in V & \text{si et seulement si} & v_i \text{ appartient à l'ensemble } V \\ \vec{v} \models \varphi_1 \wedge \cdots \wedge \varphi_m & \text{si et seulement si} & \vec{v} \models \varphi_k \text{ pour tout } k \in \{1, \ldots, m\} \\ \vec{v} \models \varphi_1 \vee \cdots \vee \varphi_m & \text{si et seulement si} & \vec{v} \models \varphi_k \text{ pour un } k \in \{1, \ldots, m\} \end{array}$$

On étend ensuite cette relation de satisfaction au cadre probabiliste de la manière suivante. Un **modèle probabiliste** est un élément de $D(V_0) \times ... \times D(V_{N-1})$. Un tel modèle associe une distribution de probabilité plutôt qu'une valeur déterminée à chaque caractère. On remarquera que cette modélisation suppose que les valeurs de deux caractères distincts sont probabilistes mais indépendantes l'une de l'autre. Étant donnés une formule φ et un modèle probabiliste $(d_0, ..., d_{N-1})$, on définit la probabilité que $\vec{d} = (d_0, ..., d_{N-1})$ satisfasse φ, notée $P(\vec{d} \models \varphi)$, ainsi :

$$P(\vec{d} \models \varphi) \stackrel{\text{def}}{=} \sum_{\vec{v} \in \vec{V},\, \vec{v} \models \varphi} \prod_{0 \leq i < N} d_i(v_i)$$

Question 21. Pour deux formules φ et ψ, montrer l'équivalence entre ces deux propositions :
- Pour tout modèle déterministe $\vec{v}$, $\vec{v} \models \varphi$ si et seulement si $\vec{v} \models \psi$.
- Pour tout modèle probabiliste $\vec{d}$, $P(\vec{d} \models \varphi) = P(\vec{d} \models \psi)$.

On dira que φ et ψ sont **équivalentes**, noté $\varphi \equiv \psi$, quand l'une ou l'autre des propositions précédentes est vraie.

Dans les questions qui suivent, on reprend le codage OCaml des distributions, et on représente directement les formules via le type suivant :

```
type formule =
  | Feuille of caractere * valeur list
  | Conjonction of formule list
  | Disjonction of formule list
```

On supposera de plus que les listes codant les ensembles de valeurs dans les formules atomiques sont sans doublons. On codera les modèles déterministes (resp. probabilistes) par des tableaux de N valeurs (resp. distributions). On notera enfin K le cardinal maximal des V_i :

$$\mathrm{K} \stackrel{\text{def}}{=} \max_{0 \leq i < N} |V_i|$$

Question 22. On considère l'algorithme qui, étant donnés un modèle probabiliste $\vec{d}$ et une formule φ, calcule $P(\vec{d} \models \varphi)$ en suivant naïvement sa définition. Quelle est la complexité temporelle de cet algorithme, en fonction de N, K et $|\varphi|$? Un programme OCaml détaillé n'est pas demandé, mais on veillera à en décrire les aspects nécessaires pour justifier l'analyse de complexité.

Dans la suite, nous allons élaborer une méthode de calcul de $P(\vec{d} \models \varphi)$ dont la complexité ne dépend pas de N. Cette méthode s'appuie sur des formes particulières de formules. Une **clause** est une conjonction de formules atomiques concernant des caractères différents. En reprenant les exemples précédents, $c_1 \in \{\mathsf{bleu}\} \wedge c_0 \in \{5\}$ est une clause mais pas $c_i \in \{1\} \wedge c_i \in \{2, 3\}$.

Question 23. Décrire un algorithme qui, étant donnés un modèle probabiliste $\vec{d}$ et une clause φ, calcule $P(\vec{d} \models \varphi)$ avec une complexité qui ne dépend que de K et $|\varphi|$, mais pas de N. Justifier la correction de l'algorithme et son analyse de complexité. Il n'est pas nécessaire de donner un programme OCaml détaillé.

Question 24. Étant données deux clauses φ et ψ, montrer que $\varphi \wedge \psi$ est équivalente à une clause que l'on notera $\varphi \overline{\wedge} \psi$.

Question 25. Montrer que pour toute formule φ il existe $m \geq 0$ et des clauses $\varphi_1, \ldots, \varphi_m$ telles que $\varphi \equiv \varphi_1 \vee \ldots \vee \varphi_m$.

Question 26. Décrire un algorithme qui, étant donnés un modèle probabiliste $\vec{d}$ et une formule φ, calcule $P(\vec{d} \models \varphi)$ avec une complexité temporelle (potentiellement non-polynomiale) qui ne dépend que de K et $|\varphi|$ mais pas de N.

En pratique, un tel algorithme est utile car on va avoir des modèles avec de nombreux caractères mais un nombre de valeurs possibles restreint pour chaque caractère, et l'on s'intéressera à des formules de taille limitée.

Fin du sujet.