TP3 - Algorithme ID3 en C

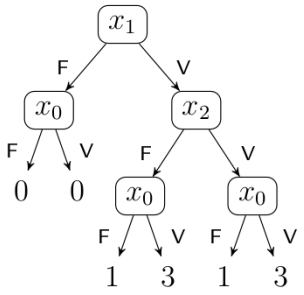
Dans ce TP, on s'intéresse à l'implémentation C de l'algorithme ID3, ainsi qu'à la manipulation d'arbres de décision. On fournit un fichier `arbre_decision.h` et `arbre_decision.c` à compléter, où sont définis les types mentionnés dans l'énoncé et certaines fonctions d'affichage.
Dans la suite, $d \in \mathbb{N}^*$ représente la dimension des données, et $K \in \mathbb{N}^*$ le nombre de classes. Le but est donc d'attribuer, à une donnée $x \in B^d$, une classe parmi $\{0, 1, ..., K-1\}$.

Arbre de décision. Un arbre de décision est un arbre binaire non vide dont :

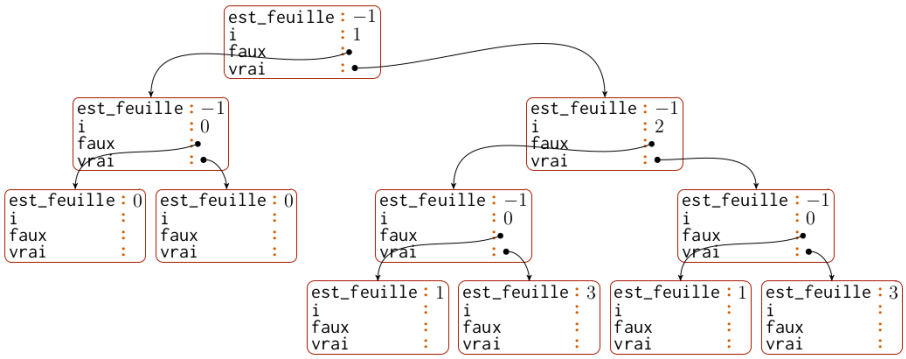
- chaque nœud interne est étiqueté par une composante $x_i \in \{x_0, x_1, ..., x_{d-1}\}$
- chaque feuille est étiquetée par une classe $c \in \{0, ..., K-1\}$

De plus, on impose que chaque composante booléenne apparaisse au plus une fois le long de chaque branche de la racine à une feuille.

Exemple : Ici, $K = 4$ et $d = 3$. L'arbre ci-dessous à gauche est un arbre de décision. À droite, on illustre la structure `C` qui permettra de représenté cet arbre.



(a) Représentation schématique



(b) Représentation en C

Un tel arbre de décision permet de représenter un classifieur pour des données dans B^d . En effet étant donné un vecteur booléen $x \in B^d$, on lit l'arbre de décision, de la racine vers les feuilles en suivant les branches correspondant aux coordonnées de x, ce qui nous conduit à une feuille nous indiquant la classe du vecteur x. Par exemple l'arbre de décision ci-dessus se lit comme suit :

- Si x_1 est F et x_0 est F alors la classe associée à (x_0, x_1, x_2) est 0.
- Si x_1 est F et x_0 est V alors la classe associée à (x_0, x_1, x_2) est 0.
- Si x_1 est V et x_2 est F et x_0 est F alors la classe associée à (x_0, x_1, x_2) est 1.
- Si x_1 est V et x_2 est F et x_0 est V alors la classe associée à (x_0, x_1, x_2) est 3.
- ....

1. Quelle est la classe de la donnée (F, F, F) d'après le classifieur représenté par l'arbre de la figure 1 ? Même question pour la donnée (F, V, F) et la donnée (V, V, F).

Représentation en C. Un nœud d'un tel arbre de décision peut être représenté en C au moyen d'une structure `struct darbre_noeud_s` contenant les 4 champs suivants :

- Un champ `int classe` valant $c \in \{0, .., K-1\}$ si le nœud est une feuille étiquetée par la classe c, et -1 si le nœud est un nœud interne.
- Trois autres champs qui ne sont significatifs que lorsque le nœud est un nœud interne
  - Un champ `int i` qui indique la variable booléenne sur laquelle le nœud réalise une disjonction.
  - Un champ `struct darbre_noeud_s* faux` contenant un pointeur vers le fils gauche du nœud.
  - Un champ `struct darbre_noeud_s* vrai` contenant un pointeur vers le fils droit du nœud.

Ainsi la définition de type `darbre` est la suivante.

```
struct darbre_noeud_s {
    int classe; /* la classe ou -1 si noeud interne */
    int i;  /* la coordonnée de disjonction */
    struct darbre_noeud_s *faux; /* le fils gauche */
    struct darbre_noeud_s *vrai; /* le fils droit */
};
typedef struct darbre_noeud_s *darbre; /* Un arbre est un pointeur
   vers sa racine */
```

Dans le fichier `arbres_decision.c`, on fournit une fonction `void affiche_arbre(darbre da)` permettant l'affichage des arbres de décision.

2. Définir les deux fonctions de création d'arbre suivantes :

- `darbre cree_feuille(int classe)`, prenant en argument une classe de $\{0, ..., K-1\}$ et retournant un arbre réduit à une feuille de cette classe.
- `darbre cree_noeud(int i, darbre faux, darbre vrai)`, prenant en arguments un entier i et deux arbres de décision `faux` et `vrai` et retournant un arbre de décision qui est un nœud interne, branchant sur x_i , dont le fils gauche est l'arbre `faux` et le fils droit est l'arbre `vrai`.

3. Définir une fonction `bool est_feuille(darbre da)` testant si l'arbre `da` est une feuille.

4. Définir une fonction `void libere_darbre(darbre da)` libérant toute la mémoire utilisée par l'arbre de décision `da`.

On appelle donnée classifiée un vecteur de d booléens $x = (x_0, ..., x_{d-1})$ auquel on a adjoint une classe (un entier dans $\{0, ..., K-1\}$). On propose le type ci-dessous :

```
struct donnee_c {
    bool* contenu; /* un tableau de booléens */
    int d; /* la taille du tableau contenu */
    int classe; /* la classe de la donnée, -1 si non définie */
};
typedef struct donnee_c donnee_c;
```

5. Définir une fonction `void lit_darbre(darbre da, donnee_c* d)` qui met à jour la classe associée à la donnée `d` grâce à la lecture de l'arbre `da`.

On s'intéresse désormais à l'implémentation de l'algorithme ID3. L'algorithme construit, à partir d'un jeu de données classifiées, un arbre de décision représentant ce jeu de données.

Pour la suite du TP, créez deux fichiers `id3.h` et `id3.c` dans lesquels vous définirez les fonctions et structures demandées.

Un jeu de données classifiées est un ensemble de données classifiées, toutes de même dimension, et ayant le même ensemble de classes possibles $\{0, ..., K-1\}$.

6. Proposer une définition de la structure `struct jeu_donnees` représentant un jeu de données classifiées afin de pouvoir définir le type suivant.

```
typedef struct jeu_donnees jeu_donnees;
```

Pour fabriquer un arbre de décision pour un jeu de données, on suit l'algorithme ID3 :

- si le jeu de données classifiées est uniforme de classe c, on crée une feuille étiquetée par c
- sinon on choisit une coordonnée i maximisant le gain d'entropie, on crée alors un nœud branchant sur la variable x_i dont le fils gauche (resp. droit) est un arbre de décision associé au jeu de données restreint aux données dont la coordonnées x_i est F (resp. V).

Profil. On appelle profil d'un jeu de données classifiées un tableau contenant `K` entiers et indiquant dans sa case d'indice `i` le nombre de données du jeu de données ayant la classe `i`.

7. Définir une fonction `void complete_profil(jeu_donnees jdc, int* profil)` prenant en arguments un jeu de données classifiées `jdc` et un tableau de taille `K` et remplissant ce tableau de sorte qu'il contienne le profil de ce jeu de données.

8. En déduire une fonction `int jdc_est_uniforme(jeu_donnees jdc)` prenant en argument un jeu de données classifiées non vide et retournant $c \in \{0, ..., K-1\}$ si le jeu de données est uniforme de classe c, et -1 s'il n'est pas uniforme.

9. Définir une fonction `double entropie(jeu_donnees jdc)` calculant l'entropie du jeu de données classifiées jdc.

10. Définir une fonction `void partitionne_jdc(jeu_donnees jdc, int i, jeu_donnees* jdc_i_faux, jeu_donnees* jdc_i_vrai)` prenant en arguments un jeu de données classifiées `jdc`, un entier `i` et deux pointeurs vers des jeux de données classifiées, et qui alloue deux nouveaux jeux de données classifiées : l'un contenant les données de jdc telles que $x_i = \text{V}$, et l'autre celles telles que $x_i = \text{F}$. Ces jeux de données classifiées doivent être stockés dans les cases mémoires pointées par `jdc_i_vrai` et `jdc_i_faux`.

11. Définir une fonction `double gain_entropie_i(jeu_donnees jdc, int i)` prenant en arguments un jeu de données classifiées `jdc` et un entier `i` et retournant le gain d'entropie du jeu de données classifiées `jdc` lorsqu'il est partitionné selon la coordonnée x_i.

Lors de l'exécution récursive de l'algorithme ID3 il est pertinent de se rappeler des coordonnées de disjonction déjà utilisées le long de la branche menant au nœud courant, afin de ne pas les considérer comme des candidats potentiels lors de futurs découpages. On représente ces coordonnées déjà utilisées au moyen d'un vecteur de booléens `utilisees` de taille `d`.

12. Définir une fonction `int gain_max(jeu_donnees jdc, bool* utilisees)` prenant en arguments un jeu de données classifiées `jdc` et un vecteur de booléens `utilisees` et calculant l'indice `i` d'une variable x_i non déjà utilisée conduisant à une partition de `jdc` de gain maximum.

13. En déduire une fonction `darbre id3(jeu_donnees_c jdc)` prenant en arguments un jeu de données classifiées `jdc` et calculant un arbre de décision pour ce jeu de données par l'algorithme ID3.