TP n°5 - Algorithme ID3

Notions abordées

- Programmation en C : `struct`, tableaux, arbres...
- Arbres de décision
- Algorithme ID3
- Implémentation d'un ensemble d'arbres à l'aide d'une table de hachage
- Partage de mémoire

Exercice 1 : Arbres de décision et algorithme ID3

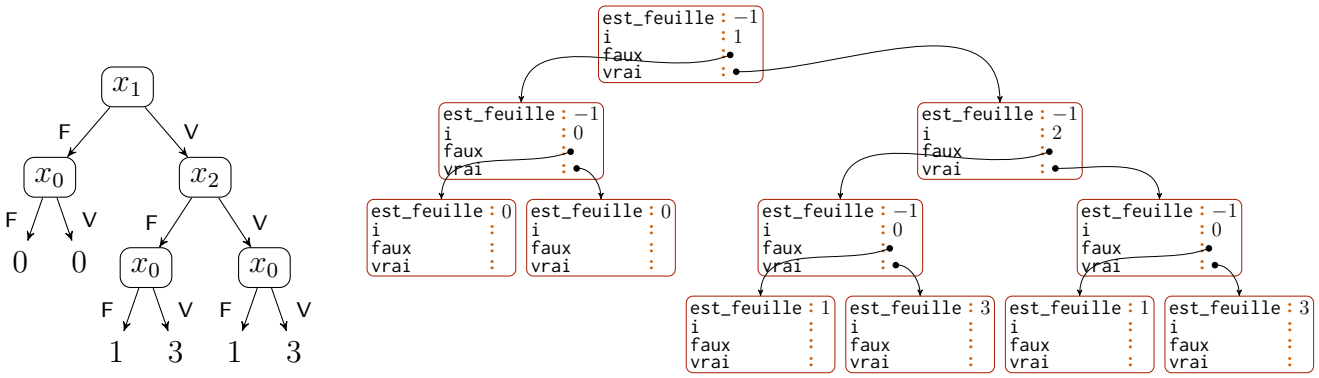
Dans cet exercice on s'intéresse à l'implémentation, en C, de l'algorithme ID3, ainsi qu'à la manipulation d'arbres de décision. La lecture de l'énoncé doit se faire accompagnée d'un survol du fichier compagnon `compagnon_arbres_decision.c`. Chaque type introduit par l'énoncé est défini dans le fichier compagnon et est accompagné d'une fonction permettant l'affichage des éléments de ce type.

Dans la suite, on fixe $n \in \mathbb{N}^\star$ la dimension des données et $K \in \mathbb{N}^\star$ représentant le nombre de classes.

Arbres de décision. Un arbre de décision est un arbre binaire non vide dont :
- chaque nœud interne est étiqueté par une variable de $\{x_0, x_1, \dots, x_{n-1}\}$;
- chaque feuille est étiquetée par une classe $c \in [\![0, K-1]\!]$.

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

Par exemple dans le cas où $K = 4$ et $n = 3$, l'arbre exemple ci-dessous à gauche (figure 1a) est un arbre de décision. Cet arbre servira d'exemple dans le reste de l'exercice.



(a) Représentation schématique (b) Représentation en C

FIGURE 1 – Deux représentations d'un arbre de décision

♣. mais pas nécessairement au moins une fois

Un tel arbre de décision permet de représenter un classifieur pour des données dans $\mathbb{B}^n$. En effet étant donné un vecteur booléen $x \in \mathbb{B}^n$, 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.
- ....

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

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.

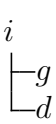
Ainsi la définition de type `darbre` est la suivante.

```c
struct darbre_noeud_s {
  int classe;                   /* la classe ou -1 si nœud 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;
```

On fournit, en figure 1b, la représentation au moyen du type `darbre` de l'arbre exemple de la figure 1a.

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

- Un nœud effectuant une disjonction sur x_i, de fils gauche g, de fils droit d est affiché sous la forme ci-contre.



- Une feuille d'étiquette c est affichée sous la forme ci-contre.

En figure 4a (page 8) on trouvera un exemple d'affichage de l'arbre exemple de l'énoncé.

1. Manipulation du type `darbre`

Q. 2 Définir, en C, 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`.

Générer des exemples. Une fois les fonctions de la question précédente implémentées, on peut utiliser les fonctions de fabrication d'arbres du fichier `compagnon_arbres_decision.c` afin d'obtenir des arbres servant d'exemples. En particulier la fonction `darbre exemple_1()` retourne l'arbre exemple de l'énoncé. Les autres fonctions exemples sont décrites en annexe page 7.

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

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

On appelle **donnée** un vecteur de booléens, représenté en C par le type ci-dessous.

```
struct bool_vec_s {
  bool* contenu;                /* un tableau de booléens */
  int   dim;                    /* la taille du tableau contenu */
};
typedef struct bool_vec_s bool_vec;
```

Q. 5 Définir, en C, une fonction `int lit_darbre(darbre da, bool_vec d)` calculant la classe associée, par l'arbre `da`, au vecteur `d`.

Q. 6 Définir, en C, une fonction `bool sont_egaux(darbre da1, darbre da2)` prenant en arguments deux arbres de décision et retournant si oui ou non ils sont égaux♣.

2. Une première compression des arbres

Deux arbres de décision sont dits **équivalents** s'ils représentent la même fonction de classificaion (de $\mathbb{B}^n$ dans $[\![0, k-1]\!]$). On dit d'un arbre de décision qu'il est **uniforme** (de classe c) dès lors que toutes ses feuilles sont étiquetées par la même classe (à savoir la classe c). Par exemple le sous-arbre gauche de l'arbre exemple ci-dessus est uniforme. On dit d'un arbre de décision qu'il est **écourté** si chacun de ses sous-arbres uniformes est réduit à une feuille. L'arbre exemple ci-dessus n'est pas écourté, le sous-arbre correspondant au cas où x_1 est F est uniforme (de classe 0) mais n'est pas réduit à une feuille. En fait dans ce cas la disjonction sur la valeur de x_0 est inutile, on peut directement conclure que la classe est 0.

Q. 7 Définir une fonction `int est_uniforme(darbre da)` prenant en argument un arbre de décision et retournant $c \in [\![0, K-1]\!]$ si l'arbre `da` est uniforme de classe c, et -1 si l'arbre n'est pas uniforme.

Q. 8 En déduire une fonction `darbre fusionne_uniforme(darbre da)` prenant en argument un arbre de décision et retournant un arbre de décision équivalent écourté. L'arbre passé en argument ne doit pas être détruit par l'appel de fonction, cette fonction doit donc allouer un nouvel arbre.

Q. 9 ★ En déduire une fonction `darbre fusionne_uniforme_modif(darbre* p_da)` prenant en argument un pointeur vers un arbre de décision et **modifiant** l'arbre de décision pointé en un arbre équivalent écourté. L'arbre passé en argument doit être modifié par l'appel de fonction. Cette fonction devra libérer la mémoire devenue inutile.

Dans l'arbre exemple de l'énoncé on remarque que la disjonction sur la variable x_2 est en fait inutile car ses deux sous-arbres (vrai et faux) sont égaux.

♣. On parle ici d'égalité structurelle et non physique : on teste si les deux arbres ont la même valeur et non s'ils sont physiquement le même arbre en mémoire.

Q. 10 ★ Donner une fonction `void supprime_disjonctions_inutiles(darbre *p_da)` prenant prenant en argument un pointeur vers un arbre de décision et **modifiant** l'arbre de décision pointé en un arbre équivalent sans disjonction inutile. L'arbre passé en argument doit être modifié par l'appel de fonction. Cette fonction devra libérer la mémoire devenue inutile.

3. Algorithme ID3

Dans cette section, on se propose d'implémenter l'algorithme ID3. L'algorithme construit, à partir d'un jeu de données classifiées, un arbre de décision représentant ce jeu de données.
On appelle **donnée classifiée** un vecteur de n booléens auquel on a adjoint une **classe** (un entier de l'intervalle entiers $[\![0, K-1]\!]$). Un **jeu de données classifiées** est un ensemble de données classifiées, toutes de même dimension, que l'on représente en C au moyen d'un tableau de données classifiées.

Q. 11 Proposer une définition de la structure `struct donnee_c_s` représentant une donnée classifiée afin de pouvoir définir le type suivant.

```c
typedef struct donnee_c_s* donnee_c;
```

Q. 12 Proposer une définition de la structure `struct jeu_donnees_c_s` représentant un jeu de données classifiées afin de pouvoir définir le type suivant.

```c
typedef struct jeu_donnees_c_s jeu_donnees_c;
```

Pour fabriquer un arbre de décision pour un jeu de données, il y a deux possibilités :
- si le jeu de données classifiées est uniforme de classe c, on crée une feuille étiquetée par c ;
- sinon on choisit une coordonnée i, 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).

Le choix de la coordonnée i est un choix glouton : on choisit la coordonnée conduisant à une partition d'entropie minimale.

Profil. On appelle **profil** d'un jeu de données classifiées un tableau K entiers (de type `int profil[K]`) indiquant dans sa case d'indice i le nombre de données du jeu de données ayant la classe i.

Q. 13 Définir une fonction `void complete_profil(jeu_donnees_c jdc, int profil[K])` 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.

Q. 14 **En déduire** une fonction `int jdc_est_uniforme(jeu_donnees_c jdc)` prenant en argument un jeu de données classifiées non vide et retournant $c \in [\![0, K-1]\!]$ si le jeu de données `jdc` est uniforme de classe c, et -1 s'il n'est pas uniforme.

Q. 15 Définir une fonction `float entropie(jeu_donnees_c jdc)` calculant l'entropie du jeu de données classifiées `jdc`.

Q. 16 Définir une fonction `void partitionne_jdc(jeu_donnees_c jdc, int i, jeu_donnees_c* jdc_i_faux, jeu_donnees_c* 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 $x_i = \mathsf{V}$, et l'autre celles telles que $x_i = \mathsf{F}$. Ces jeux de données classifiées doivent être stockés dans les cases mémoires pointées par `jdc_i_vrai` et `jdc_i_faux`.

Q. 17 Définir une fonction `float entropie_disjonction_i(jeu_donnees_c jdc, int i)` prenant en arguments un jeu de données classifiées `jdc` et un entier `i` et retournant l'entropie du jeu de données classifiées `jdc` lorsqu'il est partitionné selon la coordonnée x_i ♣

♣. D'après le cours, cette fonction retourne donc la moyenne pondérée (par les cardinaux) de l'entropie du sous-jeu de données classifiées de `jdc` des données dont la coordonnée x_i vaut F et du sous-jeu où elle vaut V.

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 n.

Q. 18 Définir une fonction `int minimiseur_entropie(jeu_donnees_c jdc, bool_vec 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` d'entropie la plus faible possible.

Q. 19 **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.

4. Pour aller plus loin : des ensembles d'arbres

La section suivante nécessite une structure de données permettant la manipulation d'un ensemble d'arbres de décisions (à travers les opérations d'ajout et de test d'appartenance). On propose d'utiliser à cet effet une table de hachage. Pour cela, on se munit d'une fonction de hachage qui associe à chaque arbre de décision un entier de l'intervalle $[\![0, H-1]\!]$ (où H est une constante de l'exercice). Un ensemble fini S d'arbres de décision est alors représenté par un tableau, de taille H, de listes d'arbres. Dans la case d'indice i de ce tableau on liste les éléments de S de hachage i. On se munit donc des types suivants.

```c
/* Type des listes d'arbres de décision */
struct darbre_list_s {
  darbre                elem ;  /* L'élément dans la cellule */
  struct darbre_list_s* next ;  /* Le suivant */
};
typedef struct darbre_list_s* darbre_list;

/* Type des ensembles d'arbres de décision */
typedef darbre_list ensemble_darbres[H];
```

Q. 20 Donner une représentation schématique de la structure de table de hachage d'arbres.

Fonction de hachage. On fixe deux constantes H (*e.g.*1049) et B (*e.g.*991), on définit alors la fonction de hachage d'un arbre de décision comme étant :

- $c \mod H$ si l'arbre est réduit à une feuille d'étiquette c ;
- $(i + s_g\, B + s_d\, B^2) \mod H$ si l'arbre est un nœud interne effectuant une disjonction sur la coordonnée x_i, que son sous-arbre gauche a pour hachage s_g et son sous-arbre droit pour hachage s_d.

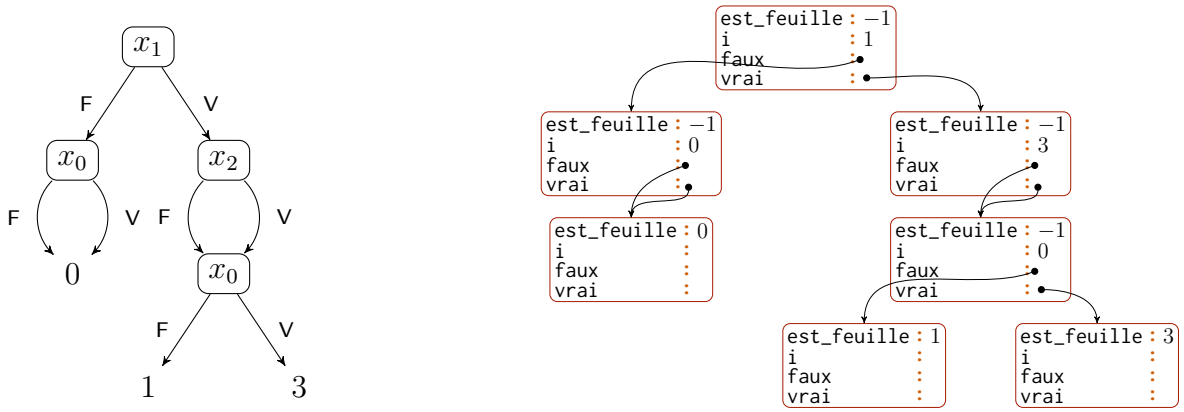
Q. 21 Définir une fonction `int hachage(darbre da)` prenant en argument un arbre de décision `da` et retournant son hachage.

Q. 22 Définir une fonction `darbre trouve_arbre(ensemble_darbres eda, darbre da)` retournant :
- un arbre se trouvant dans `eda` et égal (au sens de la **Q. 6**) à l'arbre `da` ;
- `NULL` si un tel arbre n'existe pas.

Q. 23 Définir une fonction `void ajout_arbre_tab(ensemble_darbres eda, darbre da)` ajoutant l'arbre `da` à l'ensemble d'arbres `eda`. On pourra supposer que l'arbre `da` n'est pas déjà présent dans `eda`.

5. Pour aller plus loin : mise en commun de sous-arbres

Dans l'arbre exemple de l'énoncé, on peut remarquer que certains sous-arbres apparaissent plusieurs fois : c'est le cas par exemple des sous-arbres feuilles d'étiquettes 0, 1 et 2, mais aussi des deux sous-arbres sous le nœud de disjonction x_2. La consommation mémoire d'un tel arbre de décision peut être diminuée en allouant une seule fois ces sous-arbres et les réutilisant à plusieurs endroits de l'arbre. La figure 2a propose une représentation schématique de ce partage de mémoire, tandis que la figure 2b explicite l'agencement de la mémoire C permettant un tel partage.



(a) Représentation schématique

(b) Représentation en C

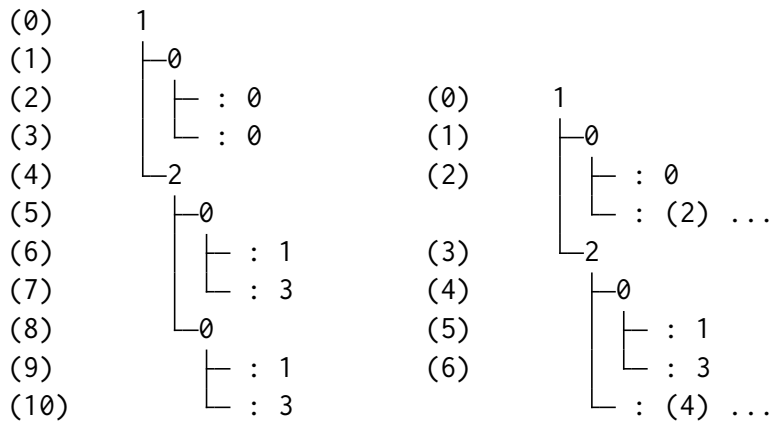
FIGURE 2 – Deux représentations d'un arbre de décision avec partage de mémoire

Affichage de mémoires partagées. Afin de rendre apparents les partages de mémoire, la fonction `void affiche_arbre_d(darbre da)` ♣ affiche au début de chaque ligne (*i.e.* pour chaque sous-arbre de l'arbre de décision) un identifiant unique entre parenthèses. Cet identifiant est ensuite réutilisé aux endroits où ce **même** sous-arbre réapparaît (par même sous-arbre, on entend **égalité physique** et non structurelle : ce sont les mêmes pointeurs).

Le code ci-contre fabrique un arbre `exemple_p` qui est identique à celui de la figure 4a, mais dans lequel plusieurs sous-arbres sont partagés en mémoire comme dans la figure 2a. On donne en figure 3 les affichages des arbres `exemple_1` et `exemple_p` produits par la fonction `affiche_arbre_d`.

```
1 darbre f0        = cree_feuille(0);
2 darbre f1        = cree_feuille(1);
3 darbre f2        = cree_feuille(2);
4 darbre n0        = cree_noeud(0, f0, f0);
5 darbre n10       = cree_noeud(0, f1, f2);
6 darbre n1        = cree_noeud(3, n10, n10);
7 darbre exemple_p = cree_noeud(1, n0, n1);
```

♣. fournie dans le fichier `compagnon_arbres_decision.c`



(a) `exemple_1` (b) `exemple_p`

FIGURE 3 – Affichages par la fonction `affiche_arbre_d`

Q. 24 Définir une fonction `void fusionne_ss_arbres_identiques(darbre *p_da)` **modifiant** l'arbre pointé par le pointeur `p_da` en mettant en commun les sous-arbres identiques. À la fin de l'exécution de la fonction l'arbre de décision `*p_da` ne devra plus contenir deux sous-arbres identiques qui ne sont pas physiquement le même arbre. On prendra garde à libérer l'espace mémoire devenu inutile.

6. Annexe

On détaille ici les fonctions de fabrication d'exemple.
Les fonctions `exemple_2`, `exemple_3`, `exemple_4` et `exemple_5` retournent des arbres de décision représentant des fonctions booléennes, autrement dit les classes sont 0 (F) ou 1 (V). Afin de décrire un tel arbre de décision il suffit donc de donner la fonction booléenne et l'ordre des variables booléennes sur les branches.

- `darbre exemple_1()` retourne l'exemple de l'énoncé (figure 1a). Cet arbre est représenté ci-dessous (figure 4a).
- `darbre exemple_2(int n)` retourne un arbre complet tel que chaque branche raisonne sur toutes les variables x_0, x_1, …, x_{n-1} ♣ et représentant la fonction booléenne :

$$(x_0, x_1, \ldots, x_{n-1}) \mapsto x_{n-1}.$$

  Un tel arbre associe donc 0 aux vecteurs booléens dont la dernière coordonnée est F et 1 aux vecteurs booléens dont la dernière coordonnée est V. À titre d'exemple on a représenté ci-dessous (figure 4b) l'arbre `exemple_2(3)`.
- `darbre exemple_3(int n)` retourne un arbre complet tel que chaque branche raisonne sur toutes les variables x_0, x_1, …, x_{n-1} ♡ et représentant la fonction booléenne :

$$(x_0, x_1, \ldots, x_{n-1}) \mapsto x_0.$$

  Un tel arbre associe donc 0 aux vecteurs booléens dont la première coordonnée est F et 1 aux vecteurs booléens dont la dernière coordonnée est V. À titre d'exemple on a représenté ci-dessous (figure 4c) l'arbre `exemple_2(3)`.

♣. et dans cet ordre
♡. et dans cet ordre

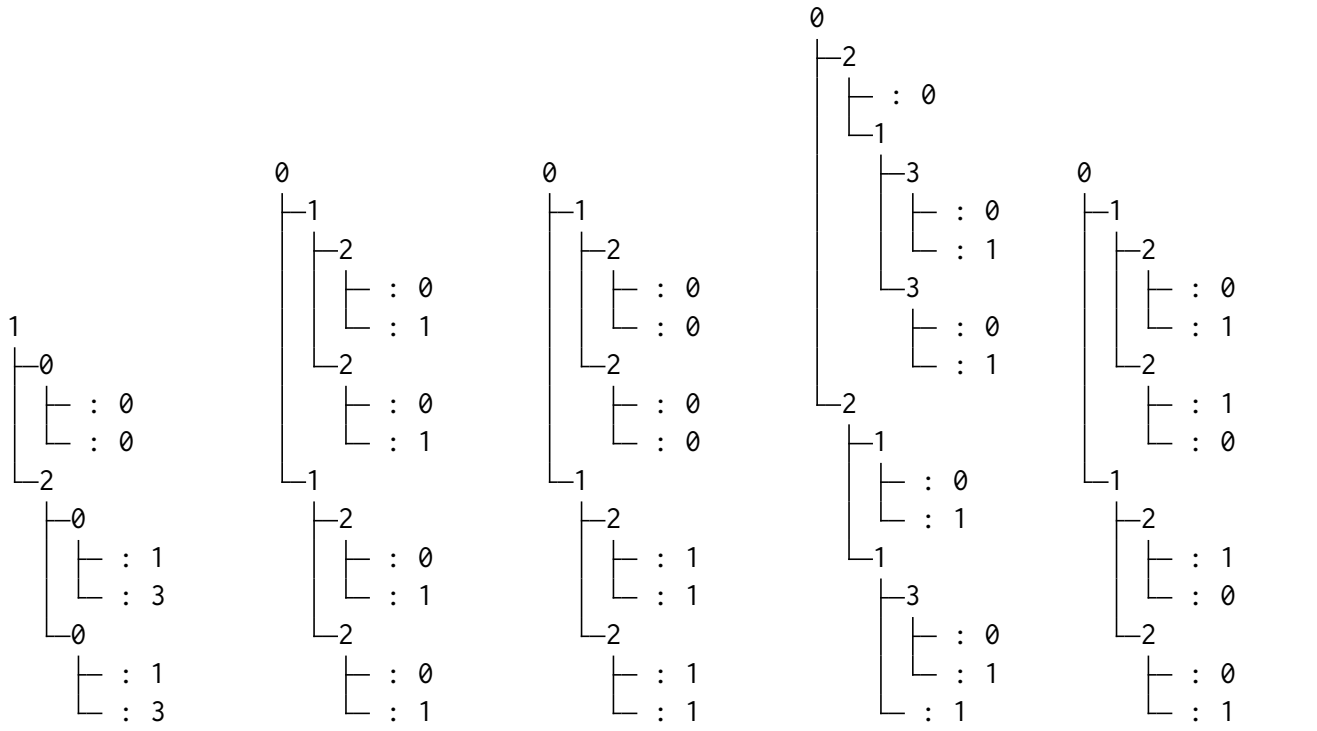
- `darbre exemple_4(int p)` retourne un arbre tel que chaque branche raisonne sur les variables $x_0, x_2, x_4, \ldots, x_{2p-2}, x_1, x_3, \ldots, x_{2p-1}$ ♣ et représentant la fonction booléenne :

$$(x_0, x_1, \ldots, x_{2p-1}) \mapsto x_0 \cdot x_1 + x_2 \cdot x_3 + \cdots + x_{2p-2} \cdot x_{2p-1}.$$

  L'arbre résultat est écourté. À titre d'exemple on a représenté ci-dessous (figure 4d) l'arbre `exemple_4(2)`.
- `darbre exemple_5(int n)` retourne un arbre complet tel que chaque branche raisonne sur toutes les variables x_0, x_1, …, x_{n-1} ♡ et représentant la fonction booléenne :

$$(x_0, x_1, \ldots, x_{n-1}) \mapsto x_0 \oplus x_1 \oplus x_2 \ldots \oplus x_{n-1}.$$

  Où $\oplus$ représente le xor. Un tel arbre associe donc 0 aux vecteurs booléens ayant un nombre pair de coordonnées V et F sinon. À titre d'exemple on a représenté ci-dessous (figure 4e) l'arbre `exemple_5(3)`.



(a) `exemple_1()` (b) `exemple_2(3)` (c) `exemple_3(3)` (d) `exemple_4(2)` (e) `exemple_5(3)`

FIGURE 4 – Les arbres exemples utilisés dans l'exercice

♣. et dans cet ordre
♡. et dans cet ordre