Concours blanc

Option informatique, deuxième année

Julien Reichert

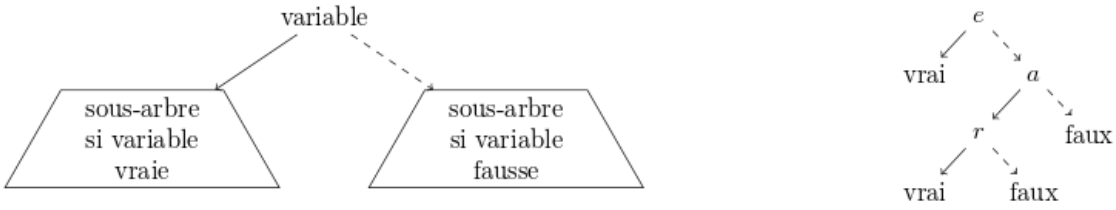
Partie 1 : Arbres de décision et diagrammes de décision

Arbres de décision

Un arbre de décision est un arbre binaire dans lequel un nœud interne est associé à une variable, parmi un ensemble V de variables et une feuille est associée à un booléen (vrai ou faux).

Si chaque variable de l'ensemble V reçoit une valeur booléenne, un tel arbre permet de prendre une décision en parcourant l'arbre : on part de la racine, quand on arrive sur un nœud interne (racine comprise), on regarde quelle est la valeur de la variable associée au nœud, si elle vaut vrai on poursuit le parcours dans le sous-arbre gauche, sinon on poursuit le parcours dans le sous-arbre droit, quand on arrive sur une feuille, le booléen associé constitue la décision.

Conventionnellement, on représente l'arête menant au sous-arbre pour le « cas vrai » en trait plein, l'arête menant au sous-arbre « cas faux » en pointillés. Schématiquement, un arbre est structuré comme indiqué ci-dessous à gauche.



Le schéma de droite ci-dessus illustre l'exemple : un module de cours est validé si l'examen est réussi (e), ou sinon, si l'étudiant a été assidu en cours (a) et qu'il réussit un examen de rattrapage (r). Cela revient à définir la validation du module par la formule logique $e \vee (a \wedge r)$.

On envisage une représentation simple d'un arbre de décision, à l'aide d'un tableau. On numérote les nœuds : la racine reçoit le numéro `0`, les autres nœuds sont numérotés arbitrairement par des entiers consécutifs à partir de `1`. On crée un tableau contenant autant de cases que de nœuds, indicé à partir de `0`. La case d'indice `i` contient soit un triplet (nom de variable, numéro du fils gauche, numéro du fils droit) si le nœud numéro `i` est un nœud interne, soit un booléen si le nœud numéro `i` est une feuille.

En Caml, on définit le type :

```
type noeud =
Feuille of bool
| Decision of string * int * int;;
```

Un arbre de décision est donc représenté par un vecteur de nœuds (type `noeud vect`).

Question 1 : Définir une variable `monAD` représentant l'arbre de décision illustré précédemment.

Dans les deux questions suivantes, on veut faire déterminer une décision en fournissant une valuation des variables, c'est-à-dire la liste des seules variables qui sont vraies dans l'évaluation.

Question 2 : Définir une fonction `eval_var` qui, étant donnés le nom d'une variable (`string`) et une liste des seules variables vraies, renvoie un booléen correspondant à la valuation de la variable indiquée.

Question 3 : Définir une fonction `eval` qui, étant donnés un arbre de décision et une liste des seules variables vraies, renvoie un booléen correspondant à la décision finale.

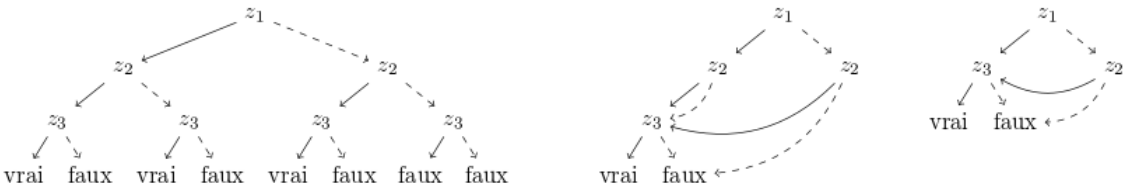
Diagrammes de décision

On souhaite compacter la représentation en mémoire des arbres de décision. Si plusieurs sous-arbres sont identiques, on n'a pas envie de les stocker plusieurs fois.

En raisonnant sur la représentation informatique des arbres de décision, on voit assez facilement une façon de procéder : si les arbres de numéros `i` et `j` sont identiques, on peut (par exemple) au niveau du parent `p` de `j` indiquer comme numéro de fils `i` au lieu de `j` et ainsi éliminer `j` de la représentation.

Ce faisant, on ne représente plus un arbre (car `i` a maintenant deux parents), mais un graphe orienté : on parle de diagrammes de décision. Néanmoins aucune connaissance particulière en théorie des graphes n'est requise pour aborder ce problème. On dira qu'il existe un arc de `p` vers `i` et on le notera $\texttt{p} \xrightarrow{b} \texttt{i}$, avec b booléen, selon que l'arc est suivi dans le cas où `p` est vrai ou faux (ce qui correspondait aux fils gauche et droit). On note de manière équivalente $\texttt{i} = \text{succ}_b(\texttt{p})$.

Exemple : l'expression $(z_1 \wedge z_3) \vee (z_2 \wedge z_3)$ admet (entre autres) les diagrammes de décision ci-dessous.



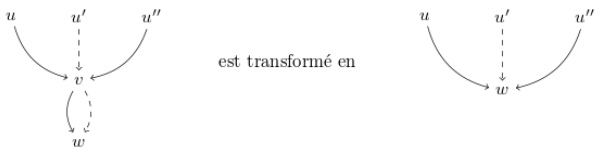
Question 4 : Créer une fonction `redirige` à trois paramètres, un diagramme ainsi que deux indices `v` et `w`, qui supprime le nœud `v` dans le graphe et transforme tous les arcs $\texttt{u} \xrightarrow{b} \texttt{v}$ en $\texttt{u} \xrightarrow{b} \texttt{w}$. Les cases du tableau qui deviennent inoccupées sont remplies avec la valeur spéciale `Vide`.

Pour ce faire en Caml on complète :

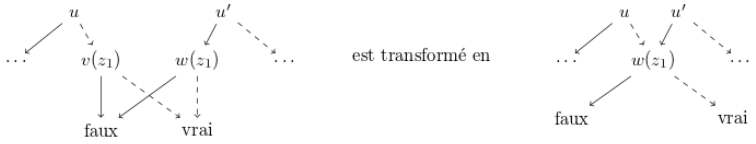
```
type noeud =
Feuille of bool
| Decision of string * int * int
| Vide;;
```

Pour transformer un arbre en diagramme sans répétition, on applique deux règles de simplification.

Élimination : Si pour un nœud `v` on a $\text{succ}_F(\texttt{v})=\text{succ}_T(\texttt{v})=\texttt{w}$ alors on élimine `v` et on transforme les arcs $\texttt{u} \xrightarrow{b} \texttt{v}$ en $\texttt{u} \xrightarrow{b} \texttt{w}$.



Isomorphisme : Soient `v` et `w` deux nœuds différents. Si ce sont des feuilles avec la même valeur de vérité ou si ce sont des nœuds internes associés à la même variable et tels que $\text{succ}_b(\texttt{v})=\text{succ}_b(\texttt{w})$ quel que soit le booléen b, alors on élimine `v` et on transforme les arcs $\texttt{u} \xrightarrow{b} \texttt{v}$ en $\texttt{u} \xrightarrow{b} \texttt{w}$.



Question 5 : Créer une fonction `trouve_elimination`, prenant en paramètre un diagramme et renvoyant l'indice d'un nœud pouvant être supprimé par élimination, s'il en existe un. Sinon, elle doit renvoyer `-1`.

Question 6 : De même, créer une fonction `trouve_isomorphisme`, prenant en paramètre un diagramme et renvoyant un couple d'indices correspondant à deux nœuds pouvant être simplifiés par isomorphisme, s'il en existe un. Sinon, elle doit renvoyer le couple `(-1,-1)`.

On dit que le diagramme est *sous forme réduite* s'il n'existe pas de nœuds différents qui correspondent à la même formule logique.

Question 7 : Prouver qu'un diagramme est sous forme réduite si, et seulement si, ni la règle d'élimination ni la règle d'isomorphisme ne peuvent lui être appliquées.

Question 8 : Créer une fonction sans résultat appelée `reduit` prenant en paramètre un diagramme, qui détecte les deux simplifications possibles, effectue les redirections correspondantes, jusqu'à ce qu'il ne soit possible de faire aucune simplification supplémentaire.

On obtient à ce stade une représentation du diagramme simplifié sous forme d'un tableau dans lequel certaines cases ne sont plus utilisées : elles sont marquées `Vide`.

Partie 2 : Tri par sélection

Soit le programme en Caml :

```
let selection s =
  let rec aux c a =
    match a with
    | [] -> (c,a)
    | t::q ->
      if (c < t) then
        let (m,r) = aux c q in
        (m,(t::r))
      else
        let (m,r) = aux t q in
        (m,(c::r))
  in
  aux (hd s) (tl s);;
```

```
let rec trier s =
  match s with
    | [] -> []
    | t::q ->
      let (m,r) = selection s in
      m :: (trier r);;
```

Soit la constante `exemple` définie et initialisée par :

```
let exemple = [3;1;4;2];;
```

Question 1 : Détailler les étapes du calcul de `trier exemple` en précisant, pour chaque appel aux fonctions `selection`, `aux` et `trier`, la valeur du paramètre et du résultat.

Soit la fonction δ calculant le symbole de Kronecker de ses paramètres ($\delta(x, y) = 1$ si $x = y$ et 0 sinon). Soit la notation $|E|_x$ pour le nombre de fois où x est dans E.

Question 2 : Soit l'entier `m`, soient les listes d'entiers `s` de taille n et `r` de taille p telles que `(m,r)` soit `selection s`, montrer que :

- Pour tout i entre 1 et n, pour s_i le i-ième élément de `s`, $\delta(s_i,\mathtt{m}) + |\mathtt{r}|_{s_i} = |\mathtt{s}|_{s_i}$.
- $n = p + 1$
- Pour tout i entre 1 et p, pour r_i le i-ième élément de `r`, $\mathtt{m} \leq r_i$.

Dans ce but, il est possible de spécifier les propriétés que doit satisfaire la fonction `aux`. Il faut alors montrer que celles-ci sont satisfaites et les exploiter ensuite.

Question 3 : Soient les listes `s` de taille m et `r` de taille n telles que `r` soit `trier s`, montrer que :

- $m = n$
- Chaque élément de `s` apparaît dans `r`, et ce autant de fois
- `r` est dans l'ordre croissant

Question 4 : Montrer que le calcul des fonctions `selection`, `aux` et `trier` se termine quelles que soient les valeurs de leurs paramètres respectant le type des fonctions.

Question 5 : Donner des exemples de valeurs du paramètre `s` de la fonction `trier` qui correspondent aux meilleur et pire cas en nombre d'appels récursifs effectués. Montrer que la complexité de la fonction `trier` en fonction du nombre n de valeurs dans les listes données en paramètre est de $\mathcal{O}(n^2)$. Cette estimation ne prend en compte que le nombre d'appels récursifs effectués.

Partie 3 : Algorithmique et programmation

Question 1 : Réaliser une structure de file de priorité en Caml. Les opérations nécessaires sur la structure sont l'insertion d'une valeur quelconque assorti d'une priorité entière, la modification de la priorité d'une valeur et l'extraction d'une valeur de priorité minimale.

Question 2 : Pour chacune des opérations écrites dans la question précédente, donner leur complexité en fonction de la taille de la structure.

Question 3 : Utiliser cette structure pour écrire l'algorithme de Dijkstra.