# Centrale Informatique MP 2013 — Corrigé

Ce corrigé est proposé par Benjamin Monmege (ENS Cachan) ; il a été relu par Charles-Pierre Astolfi (ENS Cachan) et Guillaume Batog (Professeur en CPGE).

Ce problème étudie la représentation de formules logiques à l'aide d'arbres et de diagrammes de décision. Le sujet est composé de cinq parties.

- La partie I introduit les arbres de décision, qui sont des arbres binaires permettant de représenter des formules logiques. Un algorithme d'évaluation d'un tel arbre, représenté comme un vecteur, est demandé.
- La partie II demande d'écrire une procédure compactant au maximum un arbre de décision en construisant un « diagramme de décision » (dit réduit) dont les nœuds correspondent à des formules logiques différentes.
- La partie III commence par donner une méthode directe (sans passer par les arbres de décision) construisant un diagramme de décision à partir d'une formule logique. On s'intéresse ensuite aux diagrammes de décision ordonnés, qui ont la propriété qu'une formule logique admet un unique diagramme de décision ordonné réduit, ce qui permet de décider facilement si deux formules logiques sont équivalentes, ou si l'une est une tautologie.
- La partie IV cherche à construire un circuit logique à partir d'une formule, d'abord directement, puis en utilisant les diagrammes de décision. Pour cela, on introduit et étudie des multiplexeurs, qui sont des petits circuits basiques permettant de réaliser physiquement des diagrammes de décision.
- Enfin, la partie V, très différente des précédentes, s'intéresse à la représentation des solutions de combinaisons booléennes d'équations linéaires sur les entiers grâce à des automates finis. Une construction est proposée et il est demandé de l'appliquer à un exemple d'équation linéaire.

Les parties sont indépendantes, même si les parties II et III utilisent des définitions des parties précédentes. Le sujet aborde largement les notions de logique au programme ; plus accessoirement, la partie I utilise des arbres binaires, et la partie V des automates finis. Il s'agit d'un problème relativement facile et classique : en effet, plusieurs sujets de concours se sont déjà intéressés aux diagrammes de décision, par exemple les sujets 1999 et 2012 de l'X. Il contient assez peu de questions de programmation (celles-ci sont toutes regroupées dans les deux premières parties). Quelques questions théoriques de logique sont cependant difficiles, d'autant que la formulation de l'énoncé est parfois très floue.

Téléchargé gratuitement sur www.Doc-Solus.fr .

# Indications

## Partie I

I.C Utiliser une fonction récursive auxiliaire qui parcourt le chemin de la racine à une feuille, en fonction de la valuation des variables rencontrées, que l'on calcule grâce à la fonction de la question I.B.

## Partie II

II.B Parcourir le vecteur représentant le diagramme à l'aide d'une boucle `while` à la recherche d'un nœud où la règle d'élimination peut s'appliquer. On sortira de la boucle dès qu'un tel nœud est trouvé.

II.C Procéder de même que dans la question II.B, mais en utilisant deux boucles `while` imbriquées pour parcourir l'ensemble des couples de nœuds.

II.D Attention, l'énoncé de cette question est erroné. Prouver seulement la condition nécessaire de l'assertion : si le diagramme est réduit, alors aucune des deux règles de simplification ne peut lui être appliquée. Trouver ensuite un contre-exemple à la réciproque.

## Partie III

III.B Développer la formule $(a \to b, c) \to d, e$ pour trouver sa forme normale disjonctive, la simplifier puis la factoriser afin de voir apparaître l'opérateur $(a \to \cdot, \cdot)$.

III.C Commencer par utiliser les questions III.A et III.B pour transformer toute formule logique en une formule n'utilisant que l'opérateur $(x \to \cdot, \cdot)$ avec $x$ une variable.

III.E Montrer que pour toute valuation $v$ des variables de $f$,

$$(t \to f[t=1], f[t=0])(v) = f(v)$$

III.F Expliquer comment construire un diagramme de décision ordonné à partir d'une formule logique. En appliquant les règles de simplification de la partie II, on obtient alors un diagramme de décision ordonné où ni la règle d'élimination ni la règle d'isomorphisme ne peuvent s'appliquer. Conclure en supposant que l'équivalence de la question II.D est vraie pour les diagrammes de décision ordonnés.

III.G Faire une récurrence sur le nombre $n$ de variables de la fonction booléenne.

III.H Utiliser la construction de la question III.F, et l'unicité prouvée dans la question III.G.

## Partie IV

IV.A Commencer par dénombrer les mintermes sur un ensemble à $n$ variables.

## Partie V

V.C Faire en sorte que le langage reconnu par un état $k'$ de l'automate $\mathcal{A}_{\overrightarrow{a},k}$ soit l'ensemble des mots $v \in \mathrm{A}^*$ tels que $\overrightarrow{v}$ est une solution de $\langle \overrightarrow{a} \mid \overrightarrow{x} \rangle = k'$.

V.D En partant de l'état initial de $\mathcal{A}_{\overrightarrow{a},k}$, montrer que l'on construit uniquement des états de la forme

$$\frac{k}{2^p} - \sum_{i=1}^{p} \frac{\alpha_i}{2^{p-i+1}}$$

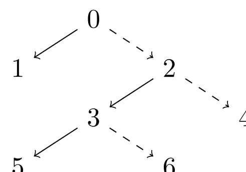
avec $p$ un entier naturel, et $(\alpha_i)_{1 \leqslant i \leqslant p}$ des entiers que l'on borne uniformément.

Téléchargé gratuitement sur www.Doc-Solus.fr .

# I. Arbres de décision

**I.A** Pour construire un arbre de décision, on peut, par exemple, créer un vecteur de type `noeud` de taille égale au nombre de nœuds de l'arbre, que l'on initialise par exemple avec des feuilles *vrai*. On remplace ensuite les nœuds un par un avec leur valeur réelle. Pour information, on a représenté à droite la numérotation choisie pour créer l'arbre.

```
let monAD = make_vect 7 (Feuille true);;
monAD.(0) <- Decision ("e",1,2);
monAD.(2) <- Decision ("a",3,4);
monAD.(3) <- Decision ("r",5,6);
monAD.(4) <- Feuille false;
monAD.(6) <- Feuille false;;
```




**I.B** On parcourt la liste `l` des seules variables vraies à la recherche de la variable `x`, grâce à une fonction récursive. Si on trouve la variable `x`, on renvoie `true` ; si on arrive à la fin de la liste `l` sans l'avoir trouvée, on renvoie `false`. La fonction `eval_var` est de type `'a -> 'a list -> bool` : en particulier, si on suppose que son premier argument est une chaîne de caractères, on obtient une fonction de type `string -> string list -> bool`.

```
let rec eval_var x l =
  match l with
  |[] -> false
  |y::l' -> (y = x) || (eval_var x l');;
```

**I.C** On utilise une fonction auxiliaire `parcours` récursive de type `int -> bool` telle que `parcours v` renvoie l'évaluation du sous-arbre de racine d'indice `v`. Si `v` est une feuille, l'évaluation est immédiate. Sinon, on calcule la valuation de la variable à la racine grâce à la fonction `eval_var` puis on évalue le sous-arbre gauche si la variable s'évalue à *vrai* et le sous-arbre droit dans le cas contraire. La fonction `parcours` est appelée à partir de la racine d'indice `0`. La fonction `eval` est de type `noeud vect -> string list -> bool`.

```
let eval arbre l =
  let rec parcours v =
    match arbre.(v) with
    |Feuille b -> b
    |Decision (x,w,w') ->
      if (eval_var x l) then parcours w else parcours w'
  in parcours 0;;
```

Téléchargé gratuitement sur www.Doc-Solus.fr .

## II. Diagrammes de décision

**II.A** La fonction `diag` commence par supprimer le nœud `v` en associant la valeur `Vide` à l'élément correspondant du vecteur codant le diagramme de décision. On parcourt ensuite ce vecteur à l'aide d'une boucle `for` à la recherche d'un nœud `u` tel qu'au moins l'un de ses sous-arbres est `v` : dans ce cas, le (ou les) sous-arbre(s) est (sont) remplacé(s) par `w`. La fonction `redirige` est de type `noeud vect -> int -> int -> unit`.

```
let redirige diag v w =
  diag.(v) <- Vide;
  for u = 0 to vect_length diag - 1 do
    match diag.(u) with
    |Decision (x,v1,v2) when v1 = v && v2 = v ->
      diag.(u) <- Decision (x,w,w)
    |Decision (x,v1,v2) when v1 = v ->
      diag.(u) <- Decision (x,w,v2)
    |Decision (x,v1,v2) when v2 = v ->
      diag.(u) <- Decision (x,v1,w)
    |_ -> ()
  done;;
```

**II.B** La fonction `trouve_elimination`, de type `noeud vect -> int`, parcourt le vecteur codant le diagramme de décision à la recherche d'un nœud `v` pouvant être éliminé. On réalise ce parcours à l'aide d'une boucle `while`. On initialise ainsi deux références `v` et `b` enregistrant l'indice courant et un booléen vrai si et seulement si un nœud pouvant être éliminé a déjà été trouvé. La référence `v` est initialisée à `0` et `b` à `false`. On sort de la boucle lorsque la référence `v` atteint la fin du vecteur ou lorsque `b` devient égal à `true`.

```
let trouve_elimination diag =
  let m = vect_length diag in
  let v = ref 0 and b = ref false in
  while !v < m && not !b do
    match diag.(!v) with
    |Decision (x,w,w') when w = w' -> b := true
    |_ -> v := !v + 1
  done;
  if !b then !v else -1;;
```

**II.C** La fonction `trouve_isomorphisme`, de type `noeud vect -> int * int`, réalise un parcours similaire à la fonction de la question II.B. À l'aide de deux boucles `while` imbriquées, l'ensemble des couples d'indices $(v, w)$, avec $v < w$, est parcouru : une boucle externe parcourt l'ensemble des indices $v$ dans l'ordre croissant, et une boucle interne parcourt l'ensemble des indices de $v+1$ au dernier dans l'ordre croissant. Pour chaque couple, on cherche si $v$ et $w$ sont deux feuilles de même valeur, ou deux nœuds vérifiant les conditions décrites dans l'énoncé.

Téléchargé gratuitement sur www.Doc-Solus.fr .