Option Informatique

CORRIGÉ

Exercice : satisfiabilité d'une formule propositionnelle.

Une formule propositionnelle est construite à l'aide de constantes propositionnelles, de variables propositionnelles et de connecteurs logiques.

Les connecteurs logiques seront notés $\neg$ (négation), $\wedge$ (conjonction), $\vee$ (disjonction). On n'en considèrera pas d'autre. Dans cette partie, on étudie le problème de satisfiabilité d'une formule.

Le problème CNF-SAT est défini de la façon suivante. Étant donné une formule sous forme normale conjonctive, admet-elle un modèle, c'est-à-dire une valuation des variables, qui rende la formule vraie ? On souhaite écrire un programme qui teste si une valuation donnée rend une telle formule vraie.

Dans cette partie, on considère que si une formule contient n variables propositionnelles, elles seront désignées par x_0, x_1, ..., x_{n-1}.

On définit le type OCaml suivant :

```
type clause = Var of int | Non of int | Ou of clause * clause
```

L'argument du constructeur `Var` correspond au numéro de la variable concernée. On notera que cette définition d'une clause ne tient pas compte de l'associativité de la disjonction.

Une formule sous forme normale conjonctive ayant `m` clauses sera implémentée par une liste de `m` termes de type `clause`.

Les tableaux seront implémentés par le module `Array` dont les éléments suivants pourront être utilisés :

- type `'a array`, notation `[| |]`
- création d'un tableau : `make : int → 'a → 'a array`
- accès à l'élément d'indice `i` du tableau `t` : `t.(i)`
- modification de l'élément placé à l'indice `i` du tableau `t` : `t.(i) <- v`
- taille du tableau : `length : 'a array -> int`

1. Donner le code OCaml correspondant à la clause $c = (x_0 \vee x_1) \vee \neg x2$.

```
let c = Ou (Ou(Var 0, Var 1), Non 2)
```

2. Donner un code OCaml permettant de définir la formule : $f = (x0 \vee x1 \vee \neg x2) \wedge (\neg x1 \vee x2)$.

```
let f = [c ; Ou (Non 1, Var 2)]
```

3. Écrire une fonction de signature `evalue_clause : clause → bool array → bool` qui prend en paramètre une clause et une valuation représentée par un tableau contenant à l'indice `i`, la valeur de vérité de la variable x_i, et renvoie la valeur de vérité de la clause.

```
let rec evalue_clause c v = match c with
|Var i -> v.(i)
|Non i -> not v.(i)
|Ou (f,g) -> evalue_clause f v || evalue clause g v
```

4. Écrire une fonction de signature `evalue_FNC : clause list → bool array -> bool` qui prend en paramètre une clause et une valuation représentée par un tableau contenant à l'indice `i`, la valeur de vérité de la variable x_i et évalue une formule donnée sous forme normale conjonctive.

```
let rec evalue_FNC fnc v = match fnc with
|[] -> true
|c::fnc2 -> evalue_clause c v && evalue_FNC fnc2 v
```

5. Quel résultat obtient-on avec la formule f et le tableau de valuations `[|false;true;true|]` ? Justifier.

Pour cette valuation on a x_0 faux, x_1 vrai et $\neg x_2$ faux donc $c = (x_0 \vee x_1 \vee \neg x_2)$ vraie, et on a $\neg x_1$ vraie et x_2 vraie donc $(\neg x_1 \vee x_2)$ vraie. Et donc on a $f = c \wedge (\neg x_1 \vee x_2)$ vraie.

6. On souhaite énumérer toutes les valuations possibles pour un nombre de variables fixé. Étant donné une valuation, on considérera que si la valeur `true` correspond à 1 et la valeur `false` correspond à 0, la valuation suivante correspond à l'ajout de 1 au nombre binaire associé. Ainsi, la valuation suivante de `[|false;true;false|]` est `[|false;true;true|]`.
On considère que la valuation suivante de `[|true;true;true|]` n'existe pas.
Écrire une fonction de signature `suivant : bool array → bool` qui prend en paramètre un tableau de booléens, et et renvoie `false` s'il n'existe pas de valuation suivante, et `true` sinon. De plus, dans ce dernier cas, cette fonction devra avoir pour effet de bord de modifier son premier argument en le transformant en la valuation suivante.

Une solution possible : j'écris d'abord une fonction qui ne retourne rien et qui modifie son argument en la valuation suivante, en provoquant une exception si c'est la dernière (ça tombe bien, ça va se faire tout seul avec un *out of bounds*).

```
let next v =
        let i = ref ((Array.length v) - 1) in
        while v.(!i) do
                v.(!i) <- false;
                decr i
                done;
        v.(!i) <- true;;

let suivant v =
        try next v; true
        with _ -> false;;
```

7. En déduire une fonction de signature `satisfiable : clause list → int → bool` qui prend en paramètre une formule en forme normale conjonctive, son nombre de variables, et renvoie `true` si il existe une valuation qui rend la formule vraie, `false` sinon.

Ceci va donner la bonne réponse grâce à l'évaluation paresseuse du `||` et du `&&` : dans la fonction auxilliaire, l'appel récursif ne se fait **que** si `fnc` n'est pas satisfaite par la valuation `v` **et que** cette valuation n'est pas la dernière, auquel cas lors de l'appel récursif, la valuation a été modifiée par l'appel de `suivant` qui a renvoyé `true`.

```
let satisfiable fnc n =
    let v = Array.make n false in
    let rec aux fnc v =
        (evalue_FNC fnc v) || (suivant v && aux fnc c) in
    aux fnc v;
```

Problème : Implémentation de dictionnaires par des ABR.

On souhaite utiliser des arbres binaires de recherches (dans la suite : ABR) pour implémenter les dictionnaires.

8. Rappeller à quelle condition un arbre binaire est un ABR (il existe deux définitions équivalentes possibles, n'en donner qu'une seule).

   Au choix :

   — Un arbre bianire est un ABR lorsque Le parcours en profondeur infixe des nœuds donne une liste triée (l'ordre sur les étiquettes étant considéré par rapport aux clés des étiquettes).

   — L'ensemble des ABR est le plus petit sous-ensemble de l'ensemble des arbres binaires tel que :
   - l'arbre vide est un ABR ;
   - un nœud n de fils gauche fg et de fils droit fd est un ABR si et seulement si
     $\begin{cases} \text{fg est un ABR ;} \\ \text{fd est un ABR ;} \\ \text{toutes les étiquettes de fg ont une clé inférieure à la clé de l'étiquette de n ;} \\ \text{toutes les étiquettes de fd ont une clé supérieure à la clé de l'étiquette de n.} \end{cases}$

A. Dictionnaires équillibrés

Une première idée est de construire un ABR le plus équillibré possible pour coder notre dictionnaire. Pour cela, il suffit de trier la liste des couples (clé, valeur) puis de stocker ses éléments dans un arbre de telle sorte que la liste triée corresponde au parcours en profondeur préfixe de l'arbre.

On définit le type `('a,'b) arbre` par

```
type ('a,'b) arbre =
| Vide
| Noeud of ('a*'b) * ('a,'b) arbre * ('a,'b) arbre;;
```

9. Étant donné un entier n, on souhaite obtenir deux entiers n_1 et n_2 tels que $\begin{cases} n = n_1 + n_2 + 1 \\ n_2 \leqslant n_1 \leqslant n_2 + 1 \end{cases}$

   Écrire une fonction `decoupage : int → int*int` telle que `decoupage n` retourne le couple (n_1, n_2).

   Par exemple `decoupage 5` devra retourner `(2,2)` mais `decoupage 6` devra retourner `(3,2)`.

   Par définition on a $n_1 = \left\lfloor \frac{n}{2} \right\rfloor$ et $n_2 = \left\lfloor \frac{n-1}{2} \right\rfloor$. On écrit donc :

   ```
   let decoupage n = (n/2,(n-1)/2);;
   ```

   Jusqu'ici, ça va.

10. Étant donnée une liste non vide `l` et un entier `k` strictement inférieur à la longueur de la liste, on peut écrire la liste sous la forme $[a_0; \ldots; a_{k-1}; a_k; a_{k+1}; \ldots; a_{n-1}]$. On souhaite alors découper cette liste en trois morceaux : la liste $[a_0; \ldots; a_{k-1}]$, l'élément a_k, et la liste $[a_{k+1}; \ldots; a_{n-1}]$.

    Écrire une fonction `partition : 'a list → int → 'a list * 'a * 'a list` telle que `partition l k` réalise le découpage explicité ci-dessus.

    Par exemple `partition 2 [1.;2.;3.;4.;5.]` devra retourner `([1.;2.],3.,[4.;5.])`.

    Il est préférable de procéder par récursivité enveloppée pour ne pas avoir à effectuer de retournement de liste.

```
let rec partition (h::t) n = match n with
  |0 -> ([],h,t)
  |_ -> let (l1,x,l2) = partition t (n-1) in (h::l1,x,l2);;
```

11. On suppose disposer de la liste des couples (clé, valeur) triée par ordre croissant des clés. En notant n le nombre de couples (clé, valeur), on peut obtenir un ABR contenant ces couples comme suit : on crée récursivement un ABR avec la sous-liste triée des n_1 premiers couples, ainsi qu'un second ABR avec la sous-liste triée des n_2 derniers couples, et on obtient l'ABR final en prenant pour racine le $(n_1+1)^{\text{ième}}$ couple (et pour fils gauches et droits les deux ABR obtenus récursivement).

    Écrire une fonction `reconstruction : ('a*'b) list → int →('a,'b) arbre` qui prend comme argument une liste `l` de couples (clé, valeur) triée par ordre croissant des clés, un entier `n` correspondant à la longueur de la liste, et qui retourne un ABR construit comme explicité précédemment.

    Cette fonction pourra utiliser les deux fonctions précédentes.

    On couple la liste juste après n_1.

```
let rec reconstruction l n = match n with
  |0 -> Vide
  |_ -> let (n1,n2) = decoupage n in
        let (l1,x,l2) = partition l n1 in
        Noeud(x,(reconstruction l1 n1),(reconstruction l2 n2));;
```

12. On suppose maintenant disposer de la liste des couples (clé, valeur) mais qu'elle n'est pas triée. Pour construire l'ABR, on commence par trier la liste, puis on utilise la fonction précédente. La stratégie choisie ici pour trier la liste est celle du tri par fusion.
    (a) Écrire une fonction `fusion : ('a*'b) list → ('a*'b) list →('a*'b) list` telle que, si l_1 et l_2 sont deux listes de couples (clé, valeur) triées par ordre croissant des clés, alors `fusion` l_1 l_2 est une liste triée par ordre croissant des clés comprenant les mêmes couples que la réunion des deux listes.
    (b) Écrire une fonction `tri : ('a*'b) list → int → ('a*'b) list` qui prend comme arguments une liste de couples (clé, valeur) ainsi que sa longueur, et qui réalise un tri par fusion de la liste. Cette fonction utilisera les fonctions `fusion`, `decoupage` et `partition` (attention, `partition` ne retourne pas un couple de listes).
    (c) Finalement, écrire une fonction `construction : ('a*'b) list →('a,'b) arbre` qui prend comme argument une liste `l` de couples (clé, valeur) non nécessairement triée, et qui retourne un ABR convenable.

        Cette fonction pourra utiliser la fonction `List.length` a condition que celle-ci soit appelée une unique fois.

    Pas de difficulté pour `fusion`, on compare les premiers éléments des couples (en fait on pourrait même comparer les couples).

```
let rec fusion l1 l2 = match (l1,l2) with
  |([],_)  -> l2
  |(_,[])  -> l1
  |((a1,b1)::t1,(a2,b2)::t2) when a1 <= a2 -> (a1,b1)::(fusion t1 l2)
  |((a1,b1)::t1,(a2,b2)::t2) -> (a2,b2)::(fusion l1 t2);;
```

Pour le tri, on couple la liste juste après n_1. Cela nous donne un triplet $(l_1, x, l2)$ et on effectue les appels récursifs sur $l1$ et $x :: l2$.

```
let rec tri l n = if n<=1 then l else begin
  let (n1,n2) = decoupage n in
  let (l1,x,l2) = partition l n1 in
  let gauche = tri l1 n1 and
      droite = tri (x::l2) (n2+1) in
  fusion gauche droite end;;
```

Enfin, pour le programme final on recolle les morceaux en calculant une unique fois au tout début la valeur de $n = List.length(l)$.

```
let construction l =
  let n = List.length l in
  let triee = (tri l n) in
  reconstruction triee n;;
```

13. L'ensemble des étapes précédentes permet d'obtenir un ABR le plus équillibré possible codant notre dictionnaire. Notons n le nombre de couples (clé, valeur). Le prétraitement est fait une fois pour toutes et les recherches dans le dictionnaires sont maintenant peu coûteuses.
    (a) Quelle est la complexité d'une recherche dans le dictionnaire ? Justifier votre réponse.
    (b) Quelle a été la complexité de l'ensemble de tout le prétraitement ? Justifier votre réponse.

    L'ABR est équilibré, sa profondeur est donc en $O(\lg(n))$. La recherche d'une clé coûte, dans le pire des cas, la profondeur de l'arbre. Elle est donc en $O(\lg(n))$.

    Le prétraitement a pour complexité :
    — le coût du calcul de la longueur de la liste : 0 (car on ne compte que les comparaisons, $O(n)$ sinon) ;
    — plus le coût du tri par fusion : $O(n \lg(n))$;
    — plus le coût de la reconstruction : 0 (car on ne compte que les comparaisons, $O(n)$ sinon).

    Total $O(n \lg(n))$.

B. Dictionnaires adaptés aux fréquences d'apparition

Dans beaucoup d'applications, on sait que les clés de recherche apparaissent avec des fréquences diverses. Par exemple, un vérificateur d'orthographe recherchera plus féquemment le mot "une" que le mot "palimpseste".

On note $k_0 < k_1 < \ldots < k_{n-1}$ (dans cet ordre) les clés de l'ABR et $f_0, f_1, \ldots, f_{n-1}$ les fréquences d'apparition de chaque clé. On note c_i le coût de la recherche de l'élément de clé k_i.
On cherche alors à à minimiser non pas $\max_{i=0..n-1} c_i$ mais $\sum_{i=0}^{n-1} f_i c_i$, quantité qu'on appelera le **poids total**.

Si on reprend l'exemple du dictionnaire `d` suivant :

```
d = {'a':6; 'b':2; 'c':3; 'd':4, 'e':9}
```

Ce dictionnaire peut être implémenté par divers ABR, comme ci-dessous :

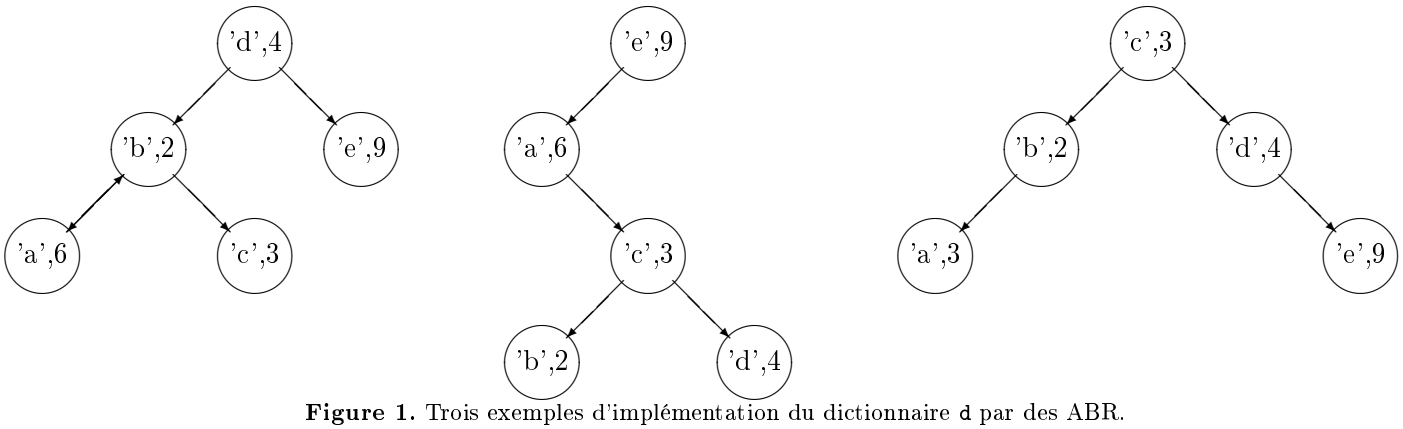

Figure 1. Trois exemples d'implémentation du dictionnaire `d` par des ABR.

On peut affecter aux lettres k_0 ='a', k_1 ='b', k_2 ='c', k_3 ='d' et k_4 ='e' leurs fréquences d'apparition dans la langue française. Pour garder des entiers, on les a multipliées ici par 10000, ce qui donne $f_0 = 815$, $f_1 = 97$, $f_2 = 315$, $f_3 = 373$, $f_4 = 1739$.

Pour ces fréquences, dans la figure 1 :

— le premier ABR a pour poids total $815 * 3 + 97 * 2 + 315 * 3 + 373 * 1 + 1739 * 2 = 7435$;

— le deuxième ABR a pour poids total $815 * 2 + 97 * 4 + 315 * 3 + 373 * 4 + 1739 * 1 = 6194$, c'est mieux ;

— le troisième ABR a pour poids total $815 * 3 + 9722 * +315 * 1 + 373 * 2 + 1739 * 3 = 10667$, c'est pire.

Notre problème est d'obtenir un ABR de clés $k_0 < k_1 < \ldots < k_{n-1}$ (dans cet ordre) ayant pour fréquences $f_0, f_1, \ldots, f_{n-1}$ mais de poids total minimal. On considère les sous-problèmes suivants : pour $0 \leqslant i \leqslant j \leqslant n-1$, trouver un sous-ABR de clés $k_i, \ldots, k_j$ (ayant pour fréquences $f_i, \ldots, f_j$) et de poids total minimal. On note alors $pt_{i,j}$ le poids total d'un tel sous-ABR.

14. Pour $i \in [|0, \ldots, n-1|]$, donner $pt_{i,i}$.

    On a $pt_{i,i} = f_i$. En effet, $pt_{i,i}$ est le poids total minimal d'un ABR n'ayant qu'un seul nœud dont la clé de l'étiquette a une féquence f_i. Mais il n'existe qu'un seul tel ABR (la feuille `Noeud((`k_i`,`e_i`),Vide,Vide)`) dont le poids total est $\sum_{k=i}^{i} f_k d_k = f_i \times 1 = f_i$, où on a noté d_p le coût d'accès à la clé k_p dans cet ABR (évidemment pour cet ABR on a nécessairement $p = i$ et $d_i = 1$).

    J'ai excessivement détaillé mais c'est pour éclairer la réponse à la question suivante.

Pour obtenir $pt_{i,j}$ avec $i < j$ on examine tous les entiers $p \in [|i, \ldots, j|]$ et on regarde parmi les ABR dont :

— la racine a pour clé k_p,

— le fils gauche est un ABR optimal de clés $k_i, \ldots, k_{p-1}$,

— le fils droit est un ABR optimal de clés $k_{p+1}, \ldots, k_j$,

lequel a un poids total minimal.

On convient de poser $pt_{i,i-1} = 0$.

15. Justifier qu'on a, pour $i \leqslant j$: $pt_{i,j} = \min_{p \in [|i,j|]} \left(pt_{i,p-1} + pt_{p+1,j} + \sum_{k=i}^{j} f_k \right)$.

    Pour $i = j$ c'est un cas particulier de la question précédente vu qu'on a $pt_{i,i-1} = pt_{i+1,i} = 0$. Plus généralement : le poids total minimal d'un ABR obtenu comme demandé par l'énoncé est le minimum pour toutes

les valeurs possibles de $p \in \{i, \dots, j\}$ des expressions de la forme $a_{i,p-1} + f_p + b_{p-1,j}$ où $a_{i,p}$ est poids induit par le fils gauche (un ABR ayant pour clés $k_i, \dots, k_{p-1}$) et $b_{p,j}$ est le poids induit par le fils droit (un ABR ayant pour clés $k_{p+1}, \dots, k_j$).

Un examen trop sommaire de la situation laisserait penser qu'on a $a_{i,p-1} = pt_{i,p-1}$ et $b_{p+1,j} = pt_{p+1,j}$, mais ce serait oublier que **tous les coûts sont augmentés de 1** puisque dans l'ABR initial tous les nœuds sont à profondeur 1 de plus que dans le fils gauche et dans le fils droit !

Notons g_r ($r \leqslant p-1$) le coût d'accès à la clé k_r dans le fils gauche et d_r ($r \geqslant p+1$) le coût d'accès à la clé k_r dans le fils droit. D'après ce qui précède on a $g_r = c_r + 1$ et $d_r = c_r + 1$.

Donc par définition : $a_{i,p-1} = \sum_{k=i}^{p-1} f_k g_k = \sum_{k=i}^{p-1} f_k(c_k+1) = \sum_{k=i}^{p-1} f_k c_k + \sum_{k=i}^{p-1} f_k = pt_{i,p-1} + \sum_{k=i}^{p-1} f_k$.

Et de même : $a_{i,p-1} = \sum_{k=p+1}^{j} f_k d_k = \sum_{k=p+1}^{j} f_k(1+c_k) = \sum_{k=p+1}^{j} f_k + \sum_{k=p+1}^{j} f_k c_k = \sum_{k=p+1}^{j} f_k + pt_{i,p-1}$.

On conclut : $a_{i,p-1} + f_p + b_{p-1,j} = pt_{i,p-1} + \sum_{k=i}^{p-1} f_k + f_p + \sum_{k=p+1}^{j} f_k + pt_{i,p-1} = pt_{i,p-1} + \sum_{k=i}^{j} f_k + pt_{i,p-1}$, et enfin $p_{i,j} = \min_{p\in\{i,\dots,j\}} \left(pt_{i,p-1} + \sum_{k=i}^{j} f_k + pt_{i,p-1} \right)$.

Remarquons que les cas $p = i$ et $p = j$ (qui correspondent au cas où la racine de l'ABR n'a qu'un seul fils) n'ont pas besoin d'être traités à part grâce à la convention $pt_{i,i-1} = 0$ (et donc $pt_{j+1,j} = 0$).

16. Déduire de la relation de récurrence précédente un algorithme de calcul de $p_{0,n-1}$ utilisant la programmation dynamique, de complexité en temps en $O(n^3)$ et de complexité en espace en $O(n^2)$. Décrire sans l'implémenter cet algorithme en justifiant les complexités indiquées.

On initialise un tableau `pt` de taille $n \times n$ (d'où une complexité en espace de $O(n^2)$).

On remplit ce tableau afin que les `pt.(i).(j)` prennent bien la valeur de $pt_{i,j}$. Pour ce faire, on peut par exemple :

- Cmmencer par la diagonale et la sous-diagonale : une même boucle for permet d'affecter tous les `pt.(i).(i)` à f_i et les `pt.(i).(i-1)` à 0 (remarque : les coefficients en dessous de cette sous-diagonale ne serviront pas). Il n'y a pas d'addition ici qui est la seule chose qu'on me demande de compter (mais même en comptant les affectations, il y en a $2n-1 = O(n)$ ce qui est négligeable vis-à-vis de la suite).
- Puis calculer colonne par colonne les `pt.(i).(j)` avec $i < j$ en remontant le long de la colonne.

  (On pourrait aussi effectuer les calculs ligne par ligne en commencant par la dernière ligne et en remontant jusqu'à la première, en parcourant chaque ligne de gauche à droite, c'est affaire de goût.)

  Le fait de travailler colonne par colonne en descendant le long de la colonne garantit que chaque nouveau calcul de coefficient ne nécessite que des coefficients déjà calculés : en effet, pour calculer $pt_{i,j}$ on va comparer toutes les quantités de la forme $pt_{i,p-1} + \sum_{k=i}^{j} f_k + pt_{p+1,j}$ et les coefficients $pt_{i,p-1}$ sont sur une colonne précédente alors que les coefficients $pt_{p+1,j}$ sont sur la même colonne mais au dessous.

  On effectue donc $\frac{n(n-1)}{2} = O(n^2)$ nouveaux calculs et chaque calcul nécessite $O(n)$ additions (et le calcul d'un minimum de n éléments, soit $O(n)$ comparaisons, mais j'écris ceci par excès de zèle puisque l'énoncé précise bien de ne compter que les additions). Ceci pourra donc se faire en imbriquant trois boucles et une complexité de $O(n^3)$.
- Une fois le tableau rempli, il n'y a plus qu'à retourner $p_{0,n-1}$.

17. Implémenter l'algorithme précédent : la fonction pourra prendre en arguments le tableau des clés **supposé trié par ordre croissant des clés** et le tableau des féquences **dans le même ordre**.

Un détail concernant l'implémentation : comme $\sum_{k=i}^{j} f_k$ ne dépend pas de p, on a $\min_{p\in\{i,\dots,j\}} \left(pt_{i,p-1} + \sum_{k=i}^{j} f_k + pt_{i,p-1}\right) = \min_{p\in\{i,\dots,j\}} \left(pt_{i,p-1} + pt_{i,p-1}\right) + \sum_{k=i}^{j} f_k$, ce qui permet de déterminer la valeur de p qui réalise le minimum avant de calculer $\sum_{k=i}^{j} f_k$. Sinon, je traduis en Caml l'algorithme précédent. L'algorithme décrit précédemment fait comme si on avait $pt_{-1,0} = pt_{n,n-1} = 0$ mais on sort ici du tableau : j'ai écrit la fonction auxilliaire `poids` pour régler le problème sans avoir à utiliser des instructions conditionnelles dans le corps de la boucle ce qui aurait alourdi le code, mais il y a certainement plus élégant.

```
let poids_total cles frequences =
  (* cles triees par ordre croissant *)
  let n = Array.length cles in
  let pt = Array.make_matrix n n (-1) in
  (* Fonction moche pour eviter les if *)
  let poids i j = if j<0 || i>=n then 0 else pt.(i).(j) in
  (* Diagonale et sous-diagonale *)
  for i = 0 to n-1 do
    pt.(i).(i) <- frequences.(i);
    if i<>0 then pt.(i).(i-1) <- 0
    done;
  (* Le reste *)
  for j=1 to n-1 do
    for i=j-1 downto 0 do
      (* Calcul de la somme des f.(k) pour k de i a j *)
      let somme = ref 0 in
      for k = i to j do
        somme := !somme + frequences.(k)
        done;
      (* Determination du meilleur entier p *)
      for p=i to j do
        let tmp = poids i (p-1) + poids (p+1) j + !somme in
        if pt.(i).(j)=(-1) || tmp < pt.(i).(j)
          then pt.(i).(j) <- tmp;
        done;
      done;
    done;
  pt.(0).(n-1);;
```

Remarque : il n'est **pas** *demandé d'adapter l'algorithme pour qu'il retourne un ABR de poids total minimal, mais seulement d'implémenter l'algorithme permettant d'obtenir le poids total.*

Je le fais quand même : on utilise un second tableau `mv` (cela double la complexité spatiale donc ne modifie pas le $O(n^2)$) qui indique pour chaque couple (i, j) quelle est la meilleure valeur p obtenue pour la racine d'un ABR de clés $k_i, \dots, k_j$. Ensuite, on reconstruit l'arbre récursivement en lisant les racines, fils gauche et fils droits des sous-arbres dans le tableau `mv`.

Code à la page suivante pour ne pas le scinder.

```
let abr_optimal cles enreg freq =
  (* cles triees par ordre croissant *)
  let n = Array.length cles in
  let pt = Array.make_matrix n n (-1) in
  let mv = Array.make_matrix n n (-1) in
  (* Fonction moche pour eviter les if *)
  let poids i j = if j<0 || i>=n then 0 else pt.(i).(j) in
  (* Diagonale et sous-diagonale *)
  for i = 0 to n-1 do
    pt.(i).(i) <- freq.(i);
    mv.(i).(i) <- i;
    if i<>0 then pt.(i).(i-1) <- 0
    done;
  (* Les autres valeurs *)
  for j=0 to n-1 do
    for i=j-1 downto 0 do
      (* Calcul de la somme des f.(k) pour k de i a j *)
      let somme = ref 0 in
      for k = i to j do
        somme := !somme + freq.(k)
        done;
      (* Determination du meilleur entier p *)
      for p=i to j do
        let tmp = poids i (p-1) + poids (p+1) j + !somme in
        if pt.(i).(j)=(-1) || tmp < pt.(i).(j) then begin
          pt.(i).(j) <- tmp;
          mv.(i).(j) <- p;
          end;
        done;
      done;
    done;
  let rec aux i j =
    if i>j then Vide
    else let p = mv.(i).(j) in
      if p<i || p>j then Vide
        else Noeud ((cles.(p),enreg.(p)),(aux i (p-1)),(aux (p+1) j)) in
  aux 0 (n-1);;
```

Remarque : on obtient que le meilleur ABR implémentant le dictionnaire d de l'introduction est le premier de l'exemple 2 qui donnait un poids total de 6194 (alors qu'il est très peu équilibré).