Option Informatique

DS05 Durée : 2h. Le 19/06/2024.

LES CALCULATRICES SONT INTERDITES

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 $c = (x_0 \vee x_1) \vee \neg x2$.

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

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.

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.

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

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.

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.

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).

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 `(n1,n2)`.
   Par exemple `decoupage 5` devra retourner `(2,2)` mais `decoupage 6` devra retourner `(3,2)`.

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.])`.

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.

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 $\texttt{l}_1$ et $\texttt{l}_2$ sont deux listes de couples (clé, valeur) triées par ordre croissant des clés, alors `fusion` $\texttt{l}_1$ $\texttt{l}_2$ 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.

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.

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 $\max\limits_{i=0..n-1} c_i$ mais $\sum\limits_{i=0}^{n-1} f_i c_i$, 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 :

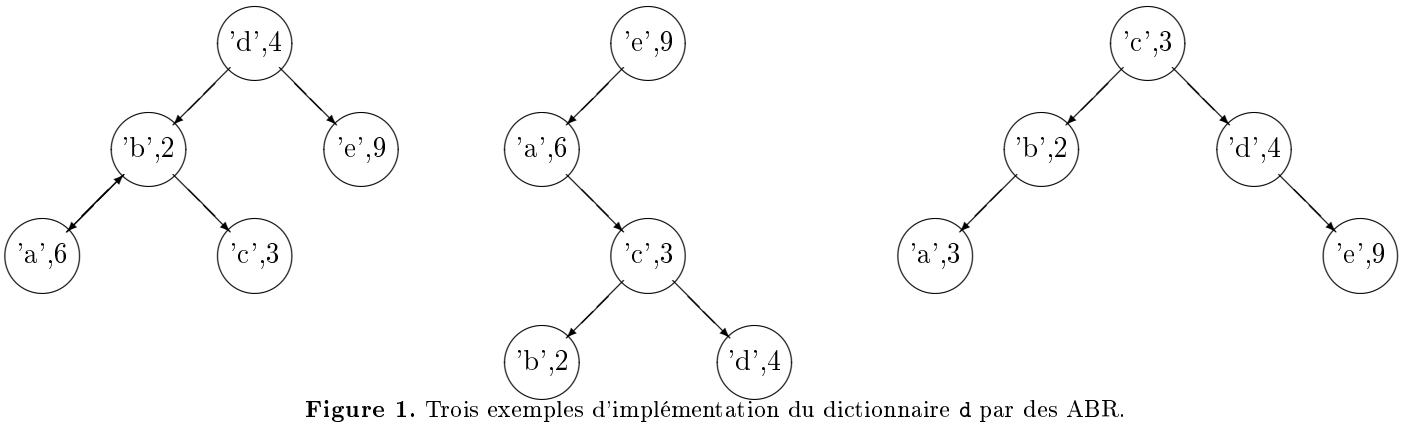

Figure 1. Trois exemples d'implémentation du dictionnaire `d` par des ABR.

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 $pt_{i,i}$.

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 $i \leqslant j$: $pt_{i,j} = \min\limits_{p \in [|i,j|]} \left(pt_{i,p-1} + pt_{p+1,j} + \sum\limits_{k=i}^{j} f_k \right)$.

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.

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**.

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.*