
Travaux pratiques de programmation

On s'attend à ce que les candidats traitent de manière au moins partielle chacune des deux parties : programmation en C et OCaml d'une part, audit de code d'autre part.

1 Programmation en C et OCaml

Dans cette partie de l'épreuve, il est demandé au candidat de concevoir, écrire et tester des programmes répondant aux questions ci-dessous, dans le langage C pour la section 1.2 et dans le langage OCaml pour la section 1.3. L'annexe à la fin de ce sujet donne des fonctions de bibliothèque pouvant être utiles.

On rappelle qu'il sera nécessaire, lors du rendu, de présenter le travail réalisé, les forces et faiblesses des approches adoptées, ainsi que d'explicitier et de commenter les jeux de tests utilisés.

1.1 Contexte

Dans ce sujet, on considère des formules de **logique propositionnelle** portant sur n variables notées $x_1, x_2, \dots, x_n$. Chacune de ces variables peut prendre la valeur **vrai** (notée $\top$) ou **faux** (notée $\perp$). Un **littéral**, noté ℓ , est une variable x_i ou sa négation notée $\neg x_i$, la négation étant définie par $\neg \top = \perp$ et $\neg \perp = \top$.

Une **clause** est une disjonction de littéraux, comme par exemple $x_1 \vee \neg x_2 \vee x_3$. Elle est vraie si l'un au moins de ses littéraux est vrai. La clause vide, ne contenant aucun littéral, n'est pas vraie. On fait dans la suite l'hypothèse que chaque clause contient **au plus une occurrence de chaque variable**, soit sous la forme x_i , soit sous la forme $\neg x_i$, de façon exclusive.

Enfin, une formule en **forme normale conjonctive** (abrégié CNF) est une conjonction de clauses. Voici un exemple de formule en CNF avec quatre clauses :

$$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \quad (1)$$

Une formule est vraie si toutes ses clauses sont vraies. La formule vide, ne contenant aucune clause, est vraie. Dans la suite, on ne considère que des formules en forme normale conjonctive.

On dit qu'une formule est **satisfiable** s'il est possible d'affecter une valeur de vérité (vrai ou faux) à ses variables qui rende la formule vraie. Ainsi, la formule (1) ci-dessus est satisfiable, car l'affectation $\{x_1 = \perp, x_2 = \perp, x_3 = \top\}$ permet de satisfaire chacune des quatre clauses. En revanche, la formule

$$(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \quad (2)$$

est **insatisfiable**, car aucune des huit façons d'affecter une valeur de vérité aux trois variables x_1, x_2, x_3 ne permet de la satisfaire.

Notre objectif est d'écrire deux programmes, utilisant deux techniques différentes, pour déterminer la satisfiabilité d'une formule donnée.

Format DIMACS. Une formule en CNF est très simplement décrite par un fichier texte dans un format appelé DIMACS. Le littéral x_i y est représenté par l'entier i et le littéral $\neg x_i$ par l'entier $-i$. Une clause est une ligne de texte contenant des littéraux séparés par un espace, et terminée par l'entier 0. Une ligne commençant par le caractère **c** est une commentaire. Enfin, les clauses sont précédées d'une ligne commençant par **p cnf** indiquant le nombre de variables et le nombre de clauses. Voici par exemple un fichier DIMACS pour la formule (1) :

```
c exemple de formule satisfiable
p cnf 3 4
1 -2 0
2 3 0
-1 -3 0
-1 -2 3 0
```

Un ensemble de formules au format DIMACS est fourni. Le répertoire **sat** contient des formules satisfiables et le répertoire **unsat** des formules insatisfiables. On trouve en particulier

- la formule (1) dans le fichier **sat/exemple-sat-1.cnf** ;
- la formule (2) dans le fichier **unsat/exemple-unsat-2.cnf** ;
- des fichiers **sat/tiroirs-sat- n .cnf** exprimant que $n + 1$ chaussettes sont rangées dans n tiroirs ;
- des fichiers **unsat/tiroirs-unsat- n .cnf** exprimant que $n + 1$ chaussettes sont rangées dans n tiroirs sans que deux chaussettes soient dans le même tiroir ;
- des fichiers **sat/reines-sat- n .cnf** exprimant que n reines sont disposées sur un échiquier $n \times n$ sans que deux d'entre elles soient en prise ;
- des fichiers **unsat/reines-unsat- n .cnf** pour $n = 2$ et $n = 3$.

Du code C et OCaml est fourni pour lire des fichiers au format DIMACS.

Format de sortie. Les deux programmes que nous nous proposons d'écrire ci-dessous prendront un nom de fichier DIMACS sur la ligne de commande et indiqueront le caractère satisfiable ou non avec une première ligne contenant le mot **SAT** ou le mot **UNSAT**. Dans le premier cas, une affectation des variables permettant de satisfaire la formule sera affichée sur une seconde ligne, dans un format similaire à celui d'une clause. Ainsi, on s'attend à une exécution de la forme suivante

```
$ ./mon-programme sat/exemple-sat-1.cnf
SAT
-1 -2 3 0
```

signifiant ici que la formule est satisfiable avec l'affectation $\{x_1 = \perp, x_2 = \perp, x_3 = \top\}$. Si plusieurs affectations sont possibles, une seule est affichée, sans préférence. Si la valeur d'une variable n'a pas besoin d'être précisée, il n'est pas nécessaire que cette variable apparaisse dans la sortie. Pour une formule insatisfiable, on a une exécution de la forme suivante

```
$ ./mon-programme unsat/exemple-unsat-2.cnf
UNSAT
```

signifiant que la formule est insatisfiable.

1.2 DPLL

Cette partie doit être réalisée en C.

Dans cette partie, nous allons déterminer la satisfiabilité d'une formule en CNF avec l'algorithme de Davis–Putnam–Logemann–Loveland, dit DPLL, qui procède par rebroussement (en anglais *backtracking*). Cet algorithme sera détaillé plus loin.

On propose des types C pour la mise en œuvre de cet algorithme. Un littéral est représenté par un entier, entre $-n$ et -1 ou entre 1 et n ,

```
typedef int literal;
```

Comme dans le format DIMACS, une valeur $i > 0$ désigne le littéral x_i et une valeur $i < 0$ désigne le littéral $\neg x_{-i}$. Une formule en CNF est représentée par le type `cnf` suivant :

```
struct cnf {
    int nb_variables;
    int nb_clauses;
    literal **clauses;
    int active;
};
```

(La signification du champ `active` sera donnée plus tard.) Le champ `clauses` est un tableau contenant les clauses. Une clause est un tableau de littéraux, **dont le premier élément est le nombre de littéraux** de la clause. Ainsi, la clause $x_1 \vee \neg x_2 \vee x_3$ est représentée par le tableau

3	1	-2	3
---	---	----	---

 de taille 4.

Une affectation est un tableau de littéraux de taille $n+1$, dont la première case est ignorée. Sa case i contient l'affectation actuelle de la variable x_i . Si $x_i = \top$, alors la case i contient la valeur i . Si $x_i = \perp$, alors la case i contient la valeur $-i$. Ainsi, l'affectation $\{x_1 = \perp, x_2 = \perp, x_3 = \top\}$ est représentée par le tableau suivant :

-	-1	-2	3
---	----	----	---

Une affectation peut être **partielle**, au sens où certaines variables ne se voient pas affecter de valeur. Dans ce cas, la case correspondante de l'affectation contient la valeur 0. On note qu'une affectation satisfait le littéral ℓ si et seulement si sa case $|\ell|$ contient ℓ .

Enfin, on se donne le type `sat` et les trois valeurs suivantes pour le résultat de certaines fonctions :

```
typedef int sat;
const sat SAT = 0;
const sat UNSAT = 1;
const sat UNKNOWN = 2;
```

Question 1. Écrire une fonction `bool check(cnf *f, literal assignment[])` qui renvoie `true` si et seulement si `assignment` est une affectation qui satisfait la formule `f`.

Question 2. Écrire une fonction `sat eval_clause(literal clause[], literal assignment[])` qui détermine si une clause donnée est satisfaite par l'affectation `assignment`. Le résultat doit être `SAT` si la clause est satisfaite, `UNSAT` si la clause est insatisfaite (tous les choix de ses littéraux sont contraires) et `UNKNOWN` sinon (aucun littéral de la clause n'est satisfait et elle contient par ailleurs au moins un littéral non affecté).

L'algorithme DPLL. C'est un algorithme récursif, qui procède par rebroussement. En entrée, on a une formule F en CNF et une affectation partielle A de ses variables, **cohérente** avec F au sens où aucune clause de F ne voit tous ses littéraux contredits par A . En sortie, on a un booléen qui vaut **true** si et seulement si il est possible de compléter cette affectation partielle de manière à satisfaire F . On procède ainsi :

1. On choisit un littéral ℓ non encore affecté.
2. On ajoute ℓ à A . Si toutes les clauses de F sont maintenant satisfaites, on renvoie **true**. Sinon, on applique DPLL récursivement. En cas de succès, on renvoie **true**.
3. Sinon, on ajoute plutôt la négation de ℓ à A et on fait la même chose. En cas de nouvel échec, on renvoie **false**.

Pour la mise en œuvre de cet algorithme, on va retirer les clauses de la formule F dès lors qu'elles sont satisfaites, pour ne pas les reconsidérer inutilement. Dès lors, il faut les **restaurer** en cas d'échec dans le point 2 ou le point 3 ci-dessus. C'est là qu'on se sert du champ **active** de la structure **cnf**. Il indique que seules les **active** premières clauses du tableau **clauses** sont à considérer. Initialement, **active** vaut **nb_clauses**. Pour supprimer la clause d'indice i , il suffit de décrémenter **active** et d'échanger les deux clauses d'indices i et **active**. Pour restaurer les clauses supprimées, il suffit de redonner à **active** son ancienne valeur. **Et il est inutile de refaire les échanges !**

Question 3. Écrire une fonction `int get_decision_variable(cnf *f, literal assignment[])` qui renvoie une variable non encore affectée dans **assignment**. On pourra supposer qu'il en existe toujours au moins une.

Question 4. Écrire une fonction `void remove_clause(cnf *f, int i)` qui supprime la clause i de la formule **f**, sous l'hypothèse $0 \leq i < f->active$.

Question 5. Écrire une fonction `sat assert_literal(cnf *f, literal assignment[], literal l)` qui ajoute le littéral l à l'affectation **assignment**. Attention, ce littéral peut être de la forme x_i ou $\neg x_i$. Les clauses satisfaites sont retirées de **f**. Le résultat doit être **SAT** s'il ne reste plus aucune clause à satisfaire à l'issue du retrait, **UNSAT** si cela mène à une clause qui ne peut être satisfaite et **UNKNOWN** sinon.

Question 6. Enfin, écrire une fonction `bool dpll(cnf *f, literal assignment[])` qui implémente l'algorithme récursif DPLL et renvoie **true** si et seulement si l'affectation **assignment** a pu être complétée de façon à satisfaire **f**. Lorsque la fonction renvoie **false**, la formule et l'affectation doivent être dans le même état qu'en entrée de la fonction.

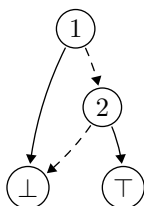
1.3 BDD

Cette partie doit être réalisée en OCaml. Les fichiers OCaml doivent être compilés avec `ocamlopt`. On pourra utiliser la commande

```
ocamlopt source.ml -o /tmp/exec && mv /tmp/exec .
```

pour accélérer le temps de compilation.

Dans cette partie, nous allons aborder le problème de la satisfiabilité différemment, en construisant une structure combinatoire appelée BDD pour *Binary Decision Diagram*. Cette structure représente une formule par un **arbre de décision**. Un BDD est un arbre binaire avec deux types de feuilles : une feuille $\top$ (pour la valeur vraie) et une feuille $\perp$ (pour la valeur fausse). Un nœud interne, noté $\langle i, n, p \rangle$, est un triplet formé du numéro i d'une variable x_i et de deux pointeurs : un **pointeur négatif** n , qui sera dessiné en pointillés, et un **pointeur positif** p , qui sera dessiné en trait plein. Voici un exemple de BDD :



Comme on le voit sur cet exemple, on a partagé la feuille $\perp$ qui apparaît deux fois dans l'arbre de décision. De manière générale, on va **systématiquement réutiliser tout sous-arbre déjà construit**, pour un partage maximal. On construit donc une structure qui s'apparente plus à un DAG qu'à un arbre.

L'interprétation d'un BDD b est une formule $F(b)$ définie de la manière suivante :

$$\begin{aligned} F(\top) &= \top \\ F(\perp) &= \perp \\ F(\langle i, n, p \rangle) &= (\neg x_i \Rightarrow F(n)) \wedge (x_i \Rightarrow F(p)) \end{aligned}$$

Le BDD donné en exemple ci-dessus représente donc la formule

$$(\neg x_1 \Rightarrow ((\neg x_2 \Rightarrow \perp) \wedge (x_2 \Rightarrow \top))) \wedge (x_1 \Rightarrow \perp)$$

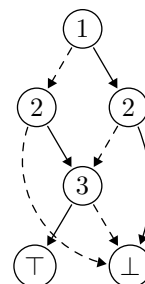
équivalente à $\neg x_1 \wedge x_2$.

Enfin, on ajoute deux contraintes à la formation de tout nœud $\langle i, n, p \rangle$ d'un BDD :

1. les variables apparaissant dans n et p sont strictement plus grandes que i ;
2. les deux formules $F(n)$ et $F(p)$ ne sont pas équivalentes.

On admet que sous ces conditions il y a unicité de représentation, dans le sens où **deux formules équivalentes sont représentées par le même BDD**.

Voici à droite un autre exemple de BDD portant sur les trois variables x_1, x_2, x_3 . Noter en particulier qu'on peut avoir plusieurs nœuds étiquetés par une même variable, comme ici x_2 . Tout **chemin** de la racine à la feuille $\top$ représente une façon de satisfaire la formule. Ici, il y a deux chemins, correspondant respectivement à $\{x_1, \neg x_2, x_3\}$ et $\{\neg x_1, x_2, x_3\}$. En particulier, une fois le BDD correspondant à une formule construit, le problème de la satisfiabilité est simple : si le BDD n'est pas identiquement égal à $\perp$, alors il existe au moins un chemin menant à $\top$ et il suffit d'en exhiber un quelconque.



Mise en œuvre en OCaml. Pour représenter les BDD, on se donne le type `bdd` suivant :

```
type variable = int
type uid = int
type bdd = Leaf of bool | Node of uid * variable * bdd * bdd
```

La valeur `Leaf false` représente la feuille $\perp$ et la valeur `Leaf true` représente la feuille $\top$. La valeur `Node( $u, i, n, p$ )` représente le nœud interne $\langle i, n, p \rangle$. L'entier u désigne un entier unique associé à chaque nœud interne distinct.

Question 7. Écrire une fonction `check: bdd -> bool` qui vérifie que les variables d'un BDD apparaissent bien dans un ordre strictement croissant.

Question 8. Écrire une fonction `mk: variable -> bdd -> bdd -> bdd` qui reçoit en paramètres une variable i , deux BDD n et p et renvoie un BDD pour la formule correspondant au nœud $\langle i, n, p \rangle$. Si n et p désignent le même BDD, alors n est renvoyé. Sinon, le nœud $\langle i, n, p \rangle$ doit être renvoyé. S'il avait déjà été construit, c'est celui-ci qui doit être renvoyé. Sinon, un nouveau nœud doit être construit avec un entier unique distinct de tous ceux utilisés dans les nœuds construits précédemment. *Indication : Utiliser une table de hachage stockant les nœuds déjà construits. La fonction de hachage peut avantageusement utiliser les entiers uniques stockés dans les BDD. La fonction d'égalité peut exploiter la propriété de partage maximal.*

Question 9. Écrire une fonction `mk_pos: variable -> bdd` qui prend un entier i et renvoie le BDD correspondant au littéral x_i . Écrire de même une fonction `mk_neg: variable -> bdd` pour le littéral $\neg x_i$.

Question 10. Écrire une fonction `mk_and: bdd -> bdd -> bdd` qui reçoit en paramètres deux BDD b_1 et b_2 et renvoie un BDD pour la formule $F(b_1) \wedge F(b_2)$. Attention à bien maintenir le caractère ordonné des BDD. *Indication : On pourra utiliser le partage maximal pour mémoriser les conjonctions de deux BDD déjà calculées.* Écrire de même une fonction `mk_or: bdd -> bdd -> bdd` pour la disjonction de deux BDD.

Question 11. Écrire une fonction `any_sat: bdd -> (variable * bool) list` qui reçoit en paramètre un BDD b différent de $\perp$ et renvoie une affectation qui satisfait la formule $F(b)$, sous la forme d'une liste de variables avec leur valeur affectée.

Question 12. Enfin, construire le BDD correspondant à la formule CNF lue dans le fichier DIMACS passé sur la ligne de commande, et afficher le résultat de sa satisfiabilité sur la sortie standard, dans le format indiqué page 2.

Toutes les affectations possibles. L'intérêt de la structure de BDD n'est pas de répondre au seul problème de la satisfiabilité, mais de capturer efficacement toutes les façons possibles de satisfaire la formule.

Question 13. Proposer une fonction `count_sat: int -> bdd -> int` qui calcule le nombre d'affectations différentes permettant de satisfaire un BDD donné. Tester sur les exemples fournis et commenter les résultats.

2 Audit de code

Dans cette partie de l'épreuve, il est demandé au candidat d'étudier un fichier source qui comporte des erreurs ou des maladresses, de qualité de code ou de fonctionnement, et il est demandé d'auditer ce fichier, c'est-à-dire :

- de comprendre et d'être capable d'expliquer le fonctionnement du code à l'oral ;
- de proposer des corrections en réécrivant certaines parties afin de corriger les erreurs ou maladresses éventuelles et de rendre le code plus clair, notamment dans une optique pédagogique ;
- de proposer des améliorations de la complexité en temps ou en mémoire, ou de la sûreté du code.

Le temps indicatif de préparation de cette partie est d'une heure. Le code ci-dessous est disponible dans votre compte, à la racine, sous les noms `~/audit1.py` et `~/audit2.c` respectivement.

2.1 Code en Python

```
1  seen = set()
2
3  def remove_dup(a):
4      for i in range(len(a)):
5          x = a[i]
6          if x in seen:
7              a[i] = a.pop()
8          else:
9              seen.add(x)
```

2.2 Code en C

```
1  int binary_search(int a[], int x, int y, int z) {
2      while (y < z) {
3          int mid = (y + z) / 2;
4          int y = a[mid];
5          if (x == y) return mid;
6          if (x < y) z = mid; else y = mid;
7      }
8      return y;
9  }
10 int remove_dup(int a[], int n) {
11     int s = 0;
12     for (int i = 0; i < n; i++) {
13         int x = a[i];
14         int p = binary_search(a, x, 0, s);
15         if (a[p] == x) continue;
16         for (int j = s; j > p; j--) a[j] = a[j-1];
17         a[p] = x;
18         s++;
19     }
20     return s;
21 }
```

3 Annexe : bibliothèque C

```
#include <stdlib.h>

int abs(int i);
    // renvoie la valeur absolue de l'entier i

void *calloc(size_t n, size_t size);
    // alloue de la mémoire pour un tableau de 'n' éléments de 'size'
    // octets chacun, et renvoie un pointeur vers cet emplacement ;
    // la mémoire allouée est initialisée avec 0
```

4 Annexe : bibliothèque OCaml

Les tables de hachage de la bibliothèque OCaml s'utilisent de la manière suivante. Pour un type de clés `key`, que l'on sait munir d'une fonction de hachage et d'une fonction d'égalité, on commence par appliquer le foncteur `Hashtbl.Make` comme ceci :

```
module H = Hashtbl.Make(struct
  type t = key
  let hash (x: key) : int = ...
  let equal (x: key) (y: key) : bool = ...
end)
```

On obtient en retour un module `H` offrant un type `'a H.t` pour des tables de hachage dont les clés sont de type `key` et les valeurs associées du type `'a`. Ce module `H` fournit notamment les trois fonctions suivantes :

```
val create: int -> 'a t
    (* 'create n' renvoie une nouvelle table de hachage, vide,
       de capacité initiale 'n' *)

val add: 'a t -> key -> 'a -> unit
    (* 'add t k v' ajoute une nouvelle entrée dans la table 't',
       associant la valeur 'v' à la clé 'k' *)

val find: 'a t -> key -> 'a
    (* 'find t k' renvoie la valeur associée à la clé 'k' dans la
       table 't', le cas échéant, et lève l'exception 'Not_found'
       si la clé 'k' est absente de la table *)
```

```
*  *
   *
```