DPLL

1 Notations

— On considérera toujours que les variables sont numérotées à partir de 1, et l'on confondra parfois la variable et son indice.
— On pourra noter $-x$ la négation d'une variable x.
— On rappelle qu'un *littéral* est soit une variable, soit la négation d'une variable. On considère que la négation d'un littéral est encore un littéral (si $l = -x$, alors $-l = x$).
— Si l est un littéral, on note $|l|$ la variable sous-jacente. Ainsi, $|x| = |-x| = x$.
— Une *occurrence* d'un littéral l est une occurrence de $|l|$. La *polarité* d'une occurrence est *positive* si c'est une occurrence de l, *négative* si c'est une occurrence de $-l$.
— Toutes les formules considérées dans le sujet sont en forme normale conjonctive. Une formule est donc un ensemble (éventuellement vide) de clauses, et une clause un ensemble (éventuellement vide) de littéraux. On supposera toujours qu'une clause contient au plus une occurrence de chaque littéral.
— Si f est une formule et l un littéral, on définit $f[l]$ comme la formule dans laquelle :
  — toutes les clauses contenant une occurrence positive de l ont été supprimées ;
  — toutes les occurrences négatives de l ont été supprimées.
  Par exemple, si $f = (x \vee y \vee z) \wedge (\neg x \vee z) \wedge (x \vee \neg y)$, alors :
  — $f[-x] = (y \vee z) \wedge (\neg y)$;
  — $f[x] = (z)$.
— Il sera ici plus pratique de remplacer le concept usuel de valuation par un autre (parfaitement équivalent). Une valuation sera la donnée d'un ensemble de littéraux $v = \{l_1, \dots, l_k\}$: si $l \in v$, alors $v(|l|) = 1$ si $l = |l|$, $v(|l|) = 0$ si $l = -|l|$.
  Par exemple, la valuation $\{-1, 3, 5\}$ est telle que $v(x_1) = 0$, $v(x_3) = 1$, $v(x_5) = 1$ et $v(x)$ non défini pour les autres variables x.

1. Pour une formule f et un littéral l quelconque, a-t-on :
   — $f[l]$ satisfiable implique f satisfiable ?
   — $f[l] \models f$?
   — f satisfiable implique $f[l]$ satisfiable ?
   — f satisfiable si et seulement si $f[l] \vee f[-l]$ satisfiable ?

2 Principe de l'algorithme

L'algorithme DPLL (*Davis–Putnam–Logemann–Loveland*) est le plus utilisé par les *SAT-solvers*. Il s'agit d'un algorithme de *backtracking* qui va simplifier petit à petit la formule en effectuant des substitutions $f[l]$. L'idée fondamentale est d'essayer au maximum d'éviter les nœuds « de branchement » (nœuds d'arité 2, où l'on va essayer $f[l]$ et $f[-l]$) en détectant des littéraux dont la valeur est « forcée ».

Propagation unitaire : si une clause c est réduite à un littéral l, alors elle ne peut être satisfaite que si le littéral est vrai. f est donc satisfiable si et seulement si $f[l]$ l'est.

Littéral pur : si un littéral l n'a que des occurrences positives, alors une valuation dans laquelle l est vrai satisfera toujours au moins autant de clauses que la même valuation dans laquelle l est faux. Ainsi, à nouveau, f est satisfiable si et seulement si $f[l]$ l'est.

Le principe est alors le suivant, pour une formule φ et une valuation partielle v :
— tant que la formule φ contient une clause unitaire $c = (l)$, on remplace φ par $\varphi[l]$ et v par $v \cup \{l\}$;
— si φ est vide, elle est satisfaite par v, si elle contient une clause vide elle est insatisfiable ;
— sinon, on cherche un littéral pur l ;
  — si l'on en trouve un, alors on remplace φ par $\varphi[l]$, v par $v \cup \{l\}$ et l'on repart au début ;
  — sinon, on choisit un littéral de branchement l' et l'on fait deux appels récursifs.
— Dans le cas où l'on branche, on ne fera bien sûr le deuxième appel que si le premier n'a pas permis de satisfaire la formule.
— De nombreux « détails » restent à régler !

3 Utilisation de MiniSat

MiniSat (`http://minisat.se/`) est un SAT-solver libre et minimaliste, qui n'est pas trop loin de l'état de l'art du milieu des années 2000. Il accepte en entrée un fichier au format DIMACS codant une formule en CNF :

```
c Ligne de commentaire
c Autre ligne de commentaire
p cnf 3 5
1 -2 3 0
2 3 0
-1 -2 -3 0
1 -3 0
1 2 0
```

- Une ligne commençant par un `c` est un commentaire et doit être ignorée. On supposera pour simplifier que ces lignes sont nécessairement au début du fichier.
- La ligne `p cnf 3 5` signifie que le problème contient 3 variables et 5 clauses. Les variables sont numérotées à partir de 1 (donc de 1 à 3 dans l'exemple ci-dessus), ce qui permet de représenter le littéral x_i par l'entier i et $\neg x_i$ par $-i$.
- Il y a ensuite une clause par ligne : les différents littéraux apparaissant dans la clause sont séparés par des espaces, et il y un 0 à la fin pour signaler la fin de la clause.
- On pourra supposer qu'une clause ne contient pas deux fois le même littéral, ni un littéral et sa négation.

Pour utiliser MiniSat, il suffit d'exécuter la commande `./minisat <input-file> <output-file>`. Ainsi, si `f.cnf` correspond à l'exemple ci-dessus, on obtient :

```
$ ./minisat f.cnf f.out
============================[MINISAT]============================
| Conflicts |     ORIGINAL     |              LEARNT              | Progress |
|           | Clauses Literals |   Limit Clauses Literals  Lit/Cl |          |
=================================================================================
|         0 |       5       12 |       1       0        0     nan |  0.000 % |
=================================================================================
restarts              : 1
conflicts             : 1              (inf /sec)
decisions             : 3              (inf /sec)
propagations          : 5              (inf /sec)
conflict literals     : 1              (0.00 % deleted)
Memory used           : 1.69 MB
CPU time              : 0 s

SATISFIABLE

$ cat f.out
SAT
1 -2 3 0
```

- Le fichier de sortie a le format suivant :
  - la première ligne est `SAT` ou `UNSAT` ;
  - si la formule est satisfiable, la suite du fichier contient un témoin de satisfiabilité, sous la forme d'une liste de littéraux terminée par un 0. Ici, le témoin fourni est $(1, 0, 1)$ (x_1 est vraie, $\neg x_2$ est vraie, x_3 est vraie).
- Si aucun fichier n'est fourni, l'entrée sera lue sur l'entrée standard.

1. Télécharger le fichier `MiniSatv1.14_linux`, le stocker dans un répertoire dans lequel vous avez les droits d'exécution, le renommer en `minisat` et le rendre exécutable. Les commandes utiles :
   - `mv` ;
   - `chmod`.
2. Utiliser `minisat` pour résoudre les différents exemples de problèmes fournis.
3. Créer manuellement un fichier codant la formule suivante, et utiliser `minisat` pour obtenir un témoin de satisfiabilité.

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

4 Mise en CNF d'un problème de Sudoku

Le problème du Sudoku peut être vu comme la 9-coloration d'un graphe dans lequel certains sommets ont une couleur fixée au départ. Le graphe contient 81 sommets, correspondant aux 81 cases de la grille numérotées ainsi :

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

Deux sommets sont adjacents si les cases correspondantes sont :
— dans la même ligne
— ou dans la même colonne
— ou dans le même carré 3×3.

1. De combien de variables aura-t-on besoin pour coder une grille comme une formule propositionnelle ?
2. Proposer une formule en CNF permettant de coder une grille de Sudoku (vierge). Combien de clauses (distinctes) cette formule contient-elle ?
3. On considère maintenant une « vraie » grille, c'est-à-dire une grille dont certaines cases sont déjà remplies (et qui admet normalement une unique solution). Comment coder une telle grille ?

Le répertoire `grilles` contient 50 grilles de Sudoku, en trois variantes :
— les fichiers `grilles-xx.txt` contiennent les grilles incomplètes (avec des 0 dans les cases vides), sous un format facilement compréhensible ;
— les fichiers `grilles_xx.cnf` contiennent les formules en CNF correspondantes, au format DIMACS ;
— les fichiers `grilles_xx.cnf.out` contiennent les grilles remplies.

4. Écrire un programme prenant en entrée le fichier renvoyé par `minisat` quand on l'appelle sur une grille de Sudoku (non contradictoire) et produisant un fichier au format des `grilles_xx.cnf.out`. On précise (information nécessaire) que les littéraux sont numérotés ainsi : $x_{i\times 81+j\times 9+c}$ pour la variable signifiant que la case de coordonnées (i, j) contient un c (avec $0 \leqslant i, j < 9$ et $1 \leqslant c \leqslant 9$).
   Vérifier que l'on retrouve bien les résultats donnés.

5 Implémentation de l'algorithme

Nous allons représenter une formule en CNF par la structure suivante :

```
typedef int litteral;

struct cnf {
    litteral **clauses;
    int nb_clauses;
    int nb_variables;
};

typedef struct cnf cnf;
```

— `clauses` est un tableau de tableaux de littéraux, de longueur au moins `nb_clauses` (il peut être plus long s'il contient des clauses qui ne sont plus actives).
— Chaque clause (chaque tableau `clauses[i]`, pour $0 \leqslant i <$ `nb_clauses`) a la forme suivante :
  — `clauses[i][0]` contient un entier positif ou nul qui indique la taille de la clause (le nombre de littéraux qu'elle contient) ;
  — les cases `clauses[i][j]` pour $1 \leqslant j \leqslant$ `clauses[i][0]` contiennent chacune un littéral, c'est-à-dire un entier k (littéral x_k) ou $-k$ (littéral $\neg x_k$) avec $1 \leqslant k \leqslant$ `nb_variables`.

On définit également le type suivant :

```
typedef int sat;

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

1. Écrire une fonction `remove_clause` qui prend en entrée un pointeur `f` vers une `cnf` et un indice `i` de clause (que l'on pourra supposer compris entre 0 et `f->nb_clauses - 1`) et supprime la clause numéro `i`. Cette fonction renverra `SAT` si la suppression de la clause rend la formule vide, `UNKNOWN` sinon.

   ```
   sat remove_clause(cnf *f, int clause_index);
   ```

   La suppression doit se faire en temps constant.

2. Écrire une fonction `remove_litteral` qui prend en entrée une clause et l'indice `i` d'un littéral et supprime ce littéral de cette clause. On pourra supposer $1 \leqslant i \leqslant$ `clause[0]` et l'on renverra `UNSAT` si la suppression rend la clause vide, `UNKNOWN` sinon.

   ```
   sat remove_litteral(litteral clause[], int lit_index);
   ```

   À nouveau, la suppression doit s'effectuer en temps constant.

3. Écrire une fonction `get_litteral_occurrence` qui prend en entrée une clause c et un littéral l, et renvoie :
   — 0 s'il n'y a pas d'occurrence de l dans c ;
   — un i tel que `clause[i]` soit une occurrence (positive ou négative) de l dans c.

   ```
   int get_litteral_occurrence(litteral clause[], litteral l);
   ```

4. Écrire une fonction `assert_litteral` ayant la spécification suivante :

   Entrées : un pointeur `f` vers une `cnf`, un tableau `valuation` de littéraux et un littéral `l`.

   Préconditions : `valuation` est de longueur `f->nb_variables + 1` et, pour $1 \leqslant i \leqslant$ `f->nb_variables`, on a `valuation[i]` qui vaut i ou $-i$ (la valeur de `valuation[0]` n'a pas d'importance).

   Effets secondaires : `f` est transformée en $f[l]$ et `valuation[abs(l)]` reçoit la valeur l.

   Valeur de retour : `SAT` si l'on remarque que la formule obtenue est une tautologie, `UNSAT` si l'on remarque qu'elle est insatisfiable, `UNKNOWN` dans les autres cas. Si l'on renvoie `SAT` ou `UNSAT`, on peut arrêter la transformation de `f` avant d'avoir traité toutes les occurrences.

   ```
   sat assert_litteral(cnf *f, litteral valuation[], litteral x);
   ```

5. Écrire une fonction `get_unit_litteral` qui prend en entrée un pointeur `f` vers une `cnf` et renvoie :
   — 0 si `f` ne contient pas de clause unitaire ;
   — un littéral l tel que f contienne une clause réduite à (l), sinon.

   ```
   litteral get_unit_litteral(cnf *f);
   ```

6. Écrire une fonction `unit_propagation` qui prend en entrée un pointeur `f` vers une `cnf` et une valuation, et effectue des propagations unitaires tant que c'est possible, en renvoyant `SAT`, `UNSAT` ou `UNKNOWN` suivant les cas.

   ```
   sat unit_propagation(cnf *f, int valuation[]);
   ```

Pour trouver un littéral pur, nous allons compter le nombre d'occurrences positives et négatives pour chaque littéral. On définit la structure suivante :

```
struct litt_occurrences {
    int pos;
    int neg;
};

typedef struct litt_occurrences litt_occurrences;
```

7. Écrire une fonction `count_occurrences` qui prend en entrée un pointeur `f` vers une `cnf` et un tableau `occs` de structures `litt_occurrences` et modifie ce tableau pour que, après l'appel, on ait (pour $0 \leqslant i \leqslant$ `f->nb_variables`) :
   — `occs[i].pos` égal au nombre d'occurrences positives de la variable `i` dans la formule `f` ;
   — `occs[i].neg` égal au nombre d'occurrences négatives.
   On pourra supposer que le tableau `occs` fourni a été alloué avec une taille suffisante, mais en revanche son contenu au moment de l'appel n'est pas spécifié.

```
void count_occurrences(cnf *f, litt_occurrences occs[]);
```

8. Écrire une fonction `get_pure_litteral` qui prend en entrée un pointeur `f` vers une `cnf` et un tableau `occs` tel que décrit ci-dessus (après appel à `count_occurrences`) et renvoie :
   — un littéral pur s'il en existe un ;
   — 0 sinon.

```
litteral get_pure_litteral(cnf *f, litt_occurrences *occs);
```

Pour trouver un littéral de branchement (si l'on n'a ni littéral unitaire, ni littéral pur), on va utiliser l'heuristique suivante :
— pour chaque variable, on compte son nombre d'occurrences positives et négatives ;
— on note n^+_{max} le nombre maximal d'occurrences positives d'une variable n^-_{max} le nombre maximal d'occurrences négatives ;
— si $n^+_{max} \geqslant n^-_{max}$, on choisit une variable ayant n^+_{max} occurrences positives et on la fixe à 1 ;
— sinon, on choisit une variable ayant n^-_{max} occurrences négatives et on la fixe à 0.

9. Écrire une fonction `get_decision_litteral` implémentant cette heuristique.

```
litteral get_decision_litteral(cnf *f, litt_occurrences occs[]);
```

10. Écrire la fonction `dpll`, qui prend en entrée :
    — un pointeur `f` vers une `cnf` ;
    — un pointeur `satisfiable` vers un `bool`
    et a le comportement suivant :
    — si la formule est satifiable, alors le booléen pointé vers `satisfiable` vaudra `true` après l'appel, et le `int*` renvoyé sera un témoin de satifiabliité ;
    — sinon, le booléen pointé vaudra `false` après l'appel, et le contenu du tableau renvoyé n'a pas d'importance.

```
int *dpll(cnf *f, bool *satisfiable);
```

Il faudra sans doute définir une fonction auxiliaire.