L3-MI / PROG
TP#9 : Solver SAT

Jérôme David

2025-11-17

Instructions de rendu

Ce TP peut être réalisé en groupe de quatre. Il servira de base pour une évaluation notée en présentiel. La notation prendra en compte la correction et qualité du code, sa documentation, ainsi que les tests que vous avez effectués. Vous devrez rendre une copie du code source pour chacune des questions (1 fichier par question) ainsi qu'un rapport expliquant comment le code a été testé. Vous illustrerez dans votre rapport le fonctionnement de votre implémentation finale sur des petites formules (une satisfiable et une insatisfiable).

Présentation

Le but de ce TP est d'implémenter un algorithme pour tester la satisfiabilité d'un formule propositionnelle donnée en forme normale conjonctive (CNF). Une formule propositionnelle est en forme normale conjonctive si elle est une conjonction ($\wedge$) de clauses où chaque clause est une disjonction de littéraux ($\vee$). Un littéral est une variable propositionnelle ou sa négation.

Une formule propositionnelle est satisfiable si il existe au moins une affectation de ses variables qui la rend vraie. Si cela n'est pas le cas, alors la formule est dite insatisfiable.

Afin de tester la satisfiabilité d'une formule, une méthode simple consiste à énumérer les affectations possible des variables et tester la formule pour chaque affectation. On peut s'arrêter dès que qu'une affectation satisfait la formule. Dans le cas ou la formule est insatisfiable il faudra tester l'ensemble des affectations possibles (2^n affectations avec n variables). Cette méthode ne passe pas à l'échelle dès que le nombre de variables devient grand ($n > 20$).

Des procédures optimisées ont été développées afin de traiter ce problème plus efficacement (même si sa complexité reste exponentielle). L'algorithme le plus connu est DPLL (Davis–Putnam–Logemann–Loveland). C'est un algorithme de backtracking qui se base entre autre sur la propagation unitaire.

Le but de ce TP est d'implémenter une variante de DPLL qui utilise le "two-watched-literals" pour la propagation unitaire.

Pour représenter une formule, nous utiliserons les structures suivantes :

```
typedef uint16_t var; // type pour les id de variables x_1: 1, x_2: 2
typedef int32_t lit; // type pour les litteraux : x_1=1, not(x_1)=-1


typedef struct Variable {
    int8_t assign; // -1 unassigned, 0 false, 1 true
} Variable;

typedef struct Clause {
    size_t size;
    lit lits[];
```

```
} Clause;

typedef struct Formula {
    size_t nvars;
    size_t nclauses;
    size_t capacity;

    Clause** clauses;
    Variable* vars; // vars[i] correspond à la variable i

    var* assigned; // historique des varriables assignee pour backtracking
    size_t nassigned;
} Formula;
```

Les variables x_i sont identifiées par leur indice $i >= 1$. L'objet de type `Variable` correspondant à la variable x_i est stocké à l'indice i dans le tableau `vars` de la structure `Formula`. On aurait pu se passer de la structure `Variable` (dans un premier temps) car elle ne contient pour l'instant que l'assignation (non assignée, vrai, faux), mais on ajoutera d'autres informations par la suite. Le tableau `assigned` de la structure `Formula` contiendra le journal des variables affectées pour pourvoir revenir en arriière.

La structure `Clause` représente sa liste des littéraux avec le tableau `lits` où la valeur absolue de chaque élément est l'identifiant de la variable et son signe repésente la polarité du littéral. Par exemple, une clause $x_1 \vee \neg x_3$ sera représentée par $[1, -3]$

1 Les structures de base

Implémentez les fonctions suivantes de création et de destruction d'une formule et de ses clauses :

1. `Formula* formula_new(size_t nvars)` qui alloue et initialise une formule qui utilise `nvars` variables. A son initialisation la formule ne contient aucune clause mais réserve de la mémoire pour en accueillir 10 clauses. Toutes les variables seront non assignées. Le tableau `assigned` sera alloué de façon à pouvoir stocker une affectation de toutes les variables, et `nassigned` sera égal à 0.
2. `Clause* clause_new(size_t size, lit lits[static size]);` qui alloue et initialise une clause (et retourne un pointeur vers celle-ci). `size` donne le nombre de littéraux et `lits` est le tableau des littéraux qui seront **recopiés** dans la clause.
3. `void formula_add_clause(Formula* f, Clause* c)` qui ajoute une clause à une formule. Cette fonction réallouera le tableau des clauses si besoin en augmentant la capacité du tableau (par réallocation) de 10 clauses.
4. `void formula_free(Formula* f)` qui libère toute la mémoire utilisée par la formule (et ses clauses).

2 Evaluation et solveur naïf

Implémentez les fonctions d'assignation et d'évaluation suivantes :

1. `bool assign(Formula* f, uint16_t var, bool value)` qui assigne `value` à la variable `var` et met à jour le tableau des variables assignées. Cette fonction retourne `false` si la variable est déjà assigné et que sa valeur est différente de celle passée en paramètre.
2. `int8_t lit_eval(Formula* f, lit l);` qui retourne 1 si le littéral est évalué à vrai, 0 si évalué à faux, -1 si non assigné.

3. `int8_t clause_eval(Formula* f, Clause* c);` qui retourne 1 si la clause est évaluée à vrai, 0 si évaluée à faux, -1 si elle n'est pas vraie et l'un des littéraux n'est pas assigné.
4. `bool formula_eval(Formula* f);` qui retourne `true` ssi l'assignation des variables rend la formule vraie (`false` si il existe au moins une clause évaluée à 0 ou -1).

En vous aidant des fonctions précédentes [1], proposez une fonction implémentant un solveur SAT naïf. Cette fonction testera itérativement toutes les possibilités et s'arrêtera dès qu'une affectation satisfaisant la formule est trouvée. Dans ce cas, la formule contiendra dans son tableau de variables une assignation qui la rend vraie. Pour cela, vous pouvez vous restreindre à un maximum de 64 variables et utiliser la représentation binaire d'un `uint64_t` pour itérer sur les différentes affectations possibles : le bit d'indice i représentera la valeur de la variable x_{i+1}. Par exemple, dans le cas d'une formule à trois variables x_1, x_2, x_3, l'entier 5 correspondra à l'assignation $[VRAI, FAUX, VRAI]$. Pour, tester toutes les possibilités, il suffit de tester toutes les valeurs de l'intervalle $[0; 8[$.

Vous relèverez les temps d'exécutions de votre solveur naïf sur différentes formules (satisfiables et non satisfiables).

3 Two-watched literals

Le principe des solveurs SAT de type DPLL est d'évaluer une formule ϕ en testant successivement pour chaque variable v : $v \wedge \phi$ ou $\neg v \wedge \phi$. L'espace de recherche peut être représenté sous la forme d'un arbre binaire où chaque noeud représente une variable. Pour chaque noeud (non feuille), certaines variables peuvent être non assignées.

Une clause unitaire est une clause qui ne contient qu'un seul littéral non assigné. En d'autres termes, c'est une clause qui n'a qu'une seule variable qui peut encore être vraie ou fausse. Une telle clause ne laisse pas de choix quant à l'affectation de la variable restante. Cela permet aux algorithmes de réduire fortement l'espace de recherche.

Par exemple, considérons la formule $(v_1 \vee v_2) \wedge (\neg v_3) \wedge (\neg v_1 \vee v_3)$. $(\neg v_3)$ est une clause unitaire et force l'affectation $v_3 = faux$. Par propagation, on obtient une autre clause unitaire : $(\neg v_1 \vee faux)$ et donc on force l'affectation $v1 = faux$, etc.

Afin d'implémenter efficacement ce mécanisme de propagation unitaire, il existe une technique appelée "two-watched literals". Elle consiste à choisir, pour chaque clause, deux littéraux à surveiller qui doivent être non assignés ou vrais. Lors de l'assignation d'une variable, les clauses qui "surveillent" cette variable doivent maintenir l'invariant "les littéraux surveillés doivent être non assignés ou au moins l'un deux est vrai" en recherchant, le cas échéant (i.e. si le littéral devient faux et que l'autre n'est pas vrai), un nouveau littéral non faux à surveiller. Si aucun littéral n'est trouvé alors la clause devient unitaire.

3.1 Enrichissement des structures

Apportez les modifications suivantes à vos structures :

1. Ajouter les indices (dans `lits`) des deux littéraux surveillés à la structure `Clause`.
2. Ajouter deux listes chainées à la structure `Variable` pour pouvoir accéder aux `Clauses` qui surveillent respectivement l'affirmation ou la négation de la variable. Les chaînons des listes seront représentés par la structure suivante :

```
struct WatchNode {
    Clause* clause;
    WatchNode* next;
    size_t lit_idx; // position du littéral surveillé dans clause->lits
};
```

1. il n'est cependant pas conseillé d'utiliser la fonction `assign`, car elle réalise des opérations inutiles pour l'algorithme naïf comme l'ajout dans le tableau des affectations et la vérification de la valeur d'affectation.

Modifiez ensuite les fonctions de création et destruction de formule en conséquence. A la création d'une clause, on peut choisir les deux premiers littéraux de la clause (si la clause ne contient qu'un littéral alors les deux indices seront 0). IL faudra, lors de l'ajout d'une clause à la formule, créer deux `WatchNode` et les insérer au bons endroits. Vous pouvez créer pour cela une fonction `void watchnode_create_and_insert(Formula* f, Clause* c, size_t watch_idx)`.

3.2 Mise à jour des littéraux surveillés

Arpès l'assignation d'une variable, l'invariant du "two-watched litteral" peut être contredit. Il faut alors essayer de mettre à jour les littéraux surveillés concernant cette variable. Par exemple, si l'on décide d'assigner $x_1 = VRAI$, toutes les clauses surveillant le littéral $\neg x_1$ doivent être vérifiées. Pour chacune de ces clauses, si l'invariant n'est plus vérifié (i.e. l'autre littéral surveillé n'est pas à évalué à vrai), il faut alors trouver un autre littéral de la clause qui soit non assigné ou vrai. On peut recenser trois cas :

— On trouve un autre littéral non assigné ou évalué à vrai : il faut changer le littéral surveillé(et penser à mettre à jour les listes chaînées).
— Il n'y a pas d'autre littéral non assigné ou évalué à vrai, mais le second littéral surveillé est non assigné : la clause devient unitaire, il faut alors assigner la variable correspondante pour que le littéral (et donc la clause) soit évalué à $VRAI$.
— Il n'y a pas d'autre littéral non assigné ou évalué à vrai, et le second littéral surveillé est évalué à $FAUX$: il y a conflit, la formule ne peut être satisfaite avec l'assignation courante.

Implémentez la fonction `bool update_watched(Formula* f, var v)` qui vérifie les clauses surveillant le littéral `v` en mettant à jour si nécessaire les littéraux surveillés des clauses ne satisfaisant plus cet invariant. La méthode retournera `true` si l'invariant est vérifié ou rétablit, `false` si ce n'est pas possible.

4 Propagation unitaire et backtracking

Le principe de l'algorithme consiste à faire un choix de variable à assigner, puis propager les mises à jour des littéraux surveillés. On parle de propagation unitaire car la mise à jour des littéraux surveillés peut engendrer des assignations (en cascade) de variables lorsque les clauses deviennent unitaires. Si la propagation échoue (i.e. engendre un conflit), il faudra alors revenir en arrière et annuler toutes les assignations faites depuis le dernier chois de variable.

4.1 Propagation

Implémentez la fonction de propagation `bool propagate(Formula* f, size_t from)` qui permet d'appliquer itérativement la mise à jour des littéraux surveillés à partir de la position `from` dans la pile des assignations jusqu'au sommet de la pile. Cette fonction retournera `true` si la propagation a pu se faire sans conflit, `false` sinon. Il faut arrêter la propagation dès le premier conflit détecté.

4.2 Backtracking

Implémenter la fonction `void backtrack(Formula* f, var v)` qui permet d'annuler l'assignation de la variable `v` et toutes les assignations faites après. Cette fonction utilisera et mettra à jour la pile des assignations (champ `assigned` de `Formula`).

5 Fonction de résolution

Vous avez maintenant toutes les briques nécessaires pour réaliser l'algorithme de résolution. L'algorithme à réaliser est le suivant. Tant que la formule n'est pas évaluée à $VRAI$, il faut :

- Choisir une variable, la sauvegarder dans une pile (une autre que celle de `Formula`), et l'assigner à $VRAI$.
- Puis tant que la propagation unitaire échoue, il faut dépiler (si possible) la dernière variable, annuler les assignations depuis cette variable, et assigner cette variable à $FAUX$. Si la pile devient vide, cela signifie que l'on a exploré l'espace de recherche sans trouver de solution, la formule est donc insatisfiable.

Implémentez votre fonction de résolution, testez sur différents exemples de formules (insatisfiable et satisfiables). Illustrez dans votre rapport le fonctionnement de votre implémentation sur des petites formules (une satisfiable et une insatisfiable).