Boolean Satisfiability and the DPLL Algorithm

Léo Leesco

Jeudi 5 février 2026

Table des matières

Si possible, veuillez consulter la version à jour.

Pour utiliser l'ensemble des fichiers fournis, il est conseillé d'utiliser la commande `why3 ide -L . dpll.mlw`. Il est nécessaire de charger les fichiers supplémentaires `filter.mlw` et `mapHigher.mlw` qui donnent certaines fonctions du programme principal.

SAT Solver

Choix de modélisation

Pour des raisons purement pratiques, je préfère utiliser les idées du sujet sans utiliser directement les implémentations proposées. Ce faisant, je "généralise" un peu les notations, en espérant qu'elles me permettent de prouver les résultats attendus :

- je ne modifie pas en place l'`assignment`, ce qui me permet d'adopter une solution purement fonctionnelle, que j'espère plus simple à prouver
- on ne restreint plus explicitement les clauses à exactement 3 littéraux :
  - ceci permet de terminer immédiatement l'algorithme lorsqu'on a une clause vide ; on doit satisfaire une union vide de litéraux, et étant

donné que l'élément neutre pour l'union est $\bot$
— lorsqu'on n'a plus de clauses à traiter, de même que dans le cas de l'union, on se trouve dans une intersection vide, qui est donc vraie.

DIMACS Format

J'adapte le parser proposé, qui me fournit un résultat extrêmement proche de celui que j'attends : je produis simplement un `list clause` au lieu d'un `array clause`.

DPLL Algorithm

L'algorithme est légèrement différent que celui proposé. Évidemment, je me fonde sur les mêmes idées.

Élimination d'un `literal` : `subst`

Dans l'algorithme "de base", on teste toutes les assignations possibles, avec quelques court-circuits. Cela signifie que, pour un `literal` donné, on va essayer de trouver un `assignment` (récursivement) qui satisfait le reste de la formule une fois substitué le `literal` par `true` ou `false`.

Dans chaque `clause`, on regarde s'il existe le `literal` qui est de la bonne positivité [1] :

— soit il est de même positivité et alors la clause est trivialement satisfaite et on peut simplement la supprimer de la liste
— soit elle est de positivité opposée, et on retire le litéral de la clause (car il contribue, dans cet `assignment`, nécessairement comme `false`)

Terminaison

Cette fonction termine trivialement (en temps linéaire). La question est de savoir si son utilisation permet de garantir que `dpll` termine : la réponse est oui car elle n'est appelée que sur des `literal`s qui appartiennent effectivement à la formule. De par son fonctionnement, il est alors évident qu'elle fait décroître strictement le nombre de `literal`s présents dans la formule résultante.

Je n'ai pas eu le temps de prouver cette propriété : il faut certainement faire une disjonction de cas en fonction de la positivité du `literal` considéré, et donc casser la logique de la fonction en deux (pour ses deux phases, qui retirent d'une part les clauses contenant le `literal` positif, et d'autre part les occurrences restantes du `literal` négatif).

1. par exemple, si on regarde x_1, x_1 est de même positivité que `true`, et de positivité opposée à `false` ; réciproquement $\neg x_1$ est de même positivité que `false` et de positivité opposée à `true`

Ceci est la raison pour laquelle les objectifs de preuve `variant` ne sont pas vérifiés pour la fonction `dpll`.

Backtracking : `dpll`

Pour récupérer l'`assignment` correspondant, on retourne récursivement l'`assignment` convenant après substitution, complémentée par la valeur du `literal` qui convient.

Naturellement, si `dpll` échoue, l'`assignment` retourné n'a pas de sens (comme attendu).

L'`assignment` correspond alors à une `list` contenant les `literal` avec la positivité calculée. Il y a plusieurs invariants associés à l'`assignment` résultant :

- il n'y a pas de doublon, c'est-à-dire qu'on n'a jamais `i` et `-i` simultanément
- l'`assignment` satisfait la `list clause` passée en argument

Pour simplifier les choses, `dpll` renvoie un `option assignment`. Assez naturellement, `None` est renvoyé qui un `assignment` ne peut pas satisfaire la `list clause` passée en argument, et `Some assignment` sinon.

Remarquons que cette méthode ne produit une valeur que pour les `literal` qui figurent réellement dans la `list clause` fournie : on peut donner une valeur arbitraire aux autres (a priori ce n'est pas attendu, et un `assignment` partiel est plus utiile en pratique).

Preuve de programme

Il faut tout d'abord prouver que l'`assignment` est `valid` : c'est le cas car on n'ajoute que des `literal`s dans l'éventuel assignment qui sont dans la formule, dont on suppose les indices bornés par `nlit`. Je n'ai pas eu le temps de prouver cela, vous pourrez voir que, dans la plupart des cas (`postcondition`), l'`assignment` obtenu est effectivement `valid`.

Ensuite, il faut prouver que l'`assignment` est `sat` vis-à-vis de la formule initiale. Pour cela, on se rend compte que l'`assignment` obtenu est en fait indépendant du `literal` considéré au moment donné et donc qu'on satisfait exactement à chaque étape un `literal` de plus.

Finalement, il faut prouver qu'il n'existe pas d'`assignment sat` si on n'a rien retourné : ceci se prouve au moment où retourne `None` (qui n'existe qu'une seule fois dans l'algorithme) puisqu'on a nécessairement une `clause` vide à cet instant. Par définition, une clause vide n'est pas satisfiable, ce qui convient.

Conclusion

Je n'ai pas eu le temps de faire toutes les preuves, mais je compte finir la preuve complète ce week-end. Si possible, veuillez consulter la version à jour.