Rapport de projet - Solveur SAT

Fait à **Paris** le **02 Décembre 2018** à destination de **Jean-Guy MAILLY**

Université Paris Descartes

MMC1E12 - **Complexité algorithmique**

Étudiants

- Jeremy MEYNADIER
- Gabriel DOISNEAU
- Sofiane TALEB

Introduction

Ce projet se place dans le cadre du module de complexité algorithmique. L'un des problèmes très représentatifs des enjeux de la complexité algorithmique est le problème du solveur SAT. Ce problème pose la question de savoir si une formule logique est satisfiable. Le nombre exponentiel de combinaisons (d'interprétations) rend l'approche par force brute rapidement inutilisable, et demande de développer un algorithme plus élaboré.

Lors de ce projet, nous avons implémenté un solveur SAT en C++ acceptant en entrée des formules logiques stockées dans des fichiers Dimacs et indiquant une solution dans le cas d'une formule satisfiable. A l'aide d'un algorithme fourni en pseudo code, nous avons pu partir d'une approche naïve puis perfectionner certains points afin d'améliorer les performances globales du programme. Ce rapport détail les choix de programmation qui ont été effectués, les améliorations apportées à l'algorithme original, et enfin les limitations de notre programme.

Choix de programmation

Architecture logicielle

Dans le cadre du cours de POO avancée, le langage C++ est utilisé. C'est pourquoi le groupe a choisi ce langage pour implémenter le solveur SAT. Ce choix nous a permis d'approfondir nos connaissances en langage bas niveau, notamment sur la manipulation des pointeurs et des itérateurs. De plus, l'utilisation d'un langage orienté objet nous a permis de tirer parti du principe d'encapsulation. Ainsi la logique est isolée et l'architecture assure une maintenabilité et une compréhension facilitée du code.

Concrètement, les formules sont représentées par des matrices, fidèles au format des données contenues dans les fichiers Dimacs. Ces matrices sont implémentées par des vecteurs d'entiers à deux dimensions (`vector<vector<int>>`), sur lesquels des boucles sont exécutées afin de parcourir les données. Le solveur SAT lui, est représenté par une classe C++ dont les méthodes publiques exposent une API permettant de déterminer si une formule est satisfiable.

Algorithme

Bien que le pseudo-code fourni dans le sujet permette une compréhension du fonctionnement, l'implémentation naïve de ce dernier s'est avérée infructueuse. En effet, la logique souffre de failles ne permettant pas de produire un solveur satisfaisant. En place, nous nous sommes inspirés du pseudo code fourni en annexe. On observe que ce dernier, toujours récursif, bénéficie d'une plus grand simplicité et clarté, en analysant les potentielles clauses vides au début afin d'en faire une condition d'arrêt. La simplification est elle fidèle au sujet.

Limitations et problèmes rencontrés

Limitations

Le programme rendu est fonctionnel quant à la question de la satisfiabilité. Hormis les limitations matérielles, les petites instances aussi bien que les grosses sont déterminée satisfiable ou non correctement la majorité du temps. Sur les 60 tests effectués, 25 ont échoués, dont 15 étaient effectués sur des grosses instances. Le solveur est donc limité mais il fonctionne bien sur les instances de taille raisonnable.

En revanche, l'affectation du modèle est quant à elle buguée. En effet, le manque de temps nous a empêché d'implémenter un test automatique, mais les instances testées à la main ne semble pas indiquer de bon modèles.

Problèmes rencontrés

Bien que la compréhension de l'algorithme soit facile dans le cas du pseudo-code fourni dans le sujet, l'implémentation de ce dernier a été chronophage. Tout d'abord, le langage choisi (C++) nous a imposé des contraintes de programmation ralentissant le développement initial. Les fonctions de backtracking et de simplification ne sont pas très longues, mais leur approche récursive rend ces dernières plus complexes à coder. La majorité du temps a cependant été dévoué à la suppression de bugs car l'équipe de projet est juniore sur le langage. Ainsi la manipulation de mémoire a été source de problèmes, en plus de l'appréhension de la logique et du comportement des itérateurs. C'est pour cette raison qu'ils ont été abandonnés au profit de boucles.

Evolutions possibles

Architecture logicielle

Bien que l'utilisation de classes ait facilité l'implémentation de l'algorithme, il serait envisageable d'en utiliser plus afin de le rendre encore plus compréhensible et flexible. Par exemple, la complexité introduite par la manipulation de tableaux à deux dimensions pourrait être encapsulée dans une nouvelle classe `Formula`, qui elle-même utiliserait des objets de type `Clause` ou `Literal`.

Le code pourrait aussi bénéficier d'une documentation plus détaillée. Aussi bien au niveau des classes et fonctions, mais aussi sur la logique sous-jacente afin de permettre à d'autres collaborateurs d'apporter leur aide.

Enfin, le programme pourrait tirer parti de tests - notamment unitaires - sur les classes et fonctions. En effet, ceux-ci permettraient de limiter l'introduction de bugs lors de la modification ou l'amélioration de l'algorithme. Bien qu'ils prennent du temps à développer ils pourraient en faire gagner à plus long terme.

Algorithme

L'algorithme implémenté est relativement naïf. Et la première amélioration possible concerne la sélection de variable. Alors que la version actuelle sélectionne une variable aléatoirement, il serait possible d'ordonner et prioriser ces dernières sur un critère déterminant. Par exemple, lister les variable par ordre décroissant de contraintes appliquées permettrait de traiter les variables les plus déterminantes en premier, et ainsi plus rapidement aboutir à une potentielle solution.

Différentes recherches ont montré beaucoup de possibilités concernant le problème SAT. L'algorithme implémenté a plusieurs fois recours à des boucles imbriquées, introduisant alors des complexités de classes $O(n^2)$ voir $O(n^2)$ dans certains cas. D'autres approches moins naïves pourraient alors réduire cette complexité afin de permettre le traitement de plus grandes instances.

Autre

Le problème de satisfiabilité étant difficile à résoudre, les algorithmes sont rapidement gourmands en ressources. Notre algorithme, développé en C++, pourrait très bien être implémenté afin de tirer parti de certaines optimisations matérielles. En effet, certaines opérations étant parallélisables, il est envisageable d'introduire du calcul distribué, ou du calcul effectué par des GPUs.

Ressources

Annexe 1 - Algorithme implémenté

```
backtracking(clauses,var) -> []
{
    if (contient_clause_vide(clauses))
        return {}
    if (vide(clause))
        return var


    next_var = choisir_sa_propchaine_var(clauses)
    a = backtracking(simplify(clauses,next_var), next_var)
    if (a == {})
        return backtracking(simplify(clauses,-next_var),-next_var)
    else
        return a.add(var);
}
```