Université de Nice Sophia-Antipolis — Année 2006/2007
Master 1 Informatique

Module Paradigmes et Langages de Programmation
Projets

Projet "Satisfiabilité dans le calcul propositionnel"
Resp. : Laurence Pierre

Descriptif

Le but de ce projet est de mettre en oeuvre et de ***comparer deux méthodes pour déterminer la satisfiabilité d'une formule du calcul propositionnel*** (problème SAT) : la méthode de Quine (ou Boole/Shannon) et la méthode de Davis et Putnam [(*)].

La *méthode de Quine* consiste simplement à construire virtuellement l'arbre d'évaluation de la formule en procédant par des évaluations partielles et des simplifications. Il est basé sur le principe suivant (dit principe d'expansion) :

$$f(x_1, x_2, \ldots, x_n) = (x_i \wedge f[x_i / 1]) \quad (\neg x_i \wedge f[x_i / 0])$$

Exemple : satisfiabilité de $(p \Rightarrow q) \Rightarrow p$?
On expanse suivant la variable p puis la variable q :

1. On considère les deux cas correspondant à p=1 et p=0
   Cas p=1 :
      2. On considère les deux cas correspondant à q=1 et q=0
   Cas p=0 :
      2. On considère les deux cas correspondant à q=1 et q=0

ce qui donne, plus précisément :

1. On considère les deux cas correspondant à p=1 et p=0 :
   Cas p=1 : on simplifie en $(1 \Rightarrow q) \quad 0$, c'est à dire $\neg q$
      2. On considère les deux cas correspondant à q=1 et q=0 :
         Cas q=1, on simplifie en 0 : *échec*
         Cas q=0, on simplifie en 1 : *succès*
   Cas p=0 : inutile de continuer puisqu'on a succès pour p=1 et q=0

On remarque que cet algorithme est applicable quelle que soit la forme de la formule à traiter.

La *méthode de Davis et Putnam* travaille sur des formules mises sous forme normale conjonctive, c'est à dire conjonction de clauses, sachant qu'une *clause* est une disjonction de littéraux (les littéraux pouvant être positifs ou négatifs). Il faut donc préalablement transformer les formules en conséquence, de la façon suivante :

- remplacement de toutes les occurrences de `exp1 <=> exp2` par
  `(exp1 => exp2)` ∧ `(exp2 => exp1)`
- remplacement de toutes les occurrences de `exp1 => exp2` par ¬ `exp1` ∨ `exp2`
- application des règles de De Morgan :
  ¬ `(exp1` ∨ `exp2)` devient `(`¬ `exp1)` ∧ `(`¬ `exp2)`

¬ (exp1 ∨ exp2) devient (¬ exp1) ∧ (¬ exp2)

- remplacement de toutes les occurrences de ¬ ¬ exp par exp
- application des règles de distributivité :

  exp1 ∧ (exp2 ∨ exp3) devient (exp1 ∧ exp2) ∨ (exp1 ∧ exp3)

  (exp1 ∨ exp2) ∧ exp3 devient (exp1 ∧ exp3) ∨ (exp2 ∧ exp3)

Une fois la forme normale conjonctive obtenue, la procédure de Davis et Putnam (avec quelques optimisations quant au choix du prochain littéral à utiliser) correspond à l'algorithme suivant :

Faire
 S'il n'y a plus de clause
 alors *succès*
 sinon si x est un mono-littéral (clause formée d'un seul littéral)
  alors on sélectionne x :
   suppression de toutes les clauses qui contiennent x,
   et suppression de ¬ x dans toutes les clauses où il apparaît
  sinon si y est un littéral pur (y apparaît dans une clause, mais pas sa négation)
   alors on sélectionne y :
    suppression de toutes les clauses qui contiennent y
   sinon sélectionner le prochain littéral z non encore utilisé
    suppression de toutes les clauses qui contiennent z,
    et suppression de ¬ z dans toutes les clauses où il apparaît,
    si on a généré la clause vide
     alors on sélectionne ¬ z :
      suppression de toutes les clauses qui contiennent ¬ z,
      et suppression de z dans toutes les clauses où il apparaît
      si on a généré la clause vide alors *échec*
tant qu'on n'a pas succès (il n'y a plus de clause) ou échec (on a généré la clause vide).

Exemple : satisfiabilité de (¬ a ∧ b) => (b ∨ c) ?
La mise sous forme normale conjonctive donne : (a ∨ b) ∧ (a ∨ c) ∧ (¬ b ∨ b) ∧ (¬ b ∨ c), c'est à dire une conjonction de 4 clauses.

1. Il n'y a pas de mono-littéral, on sélectionne a qui est un littéral pur :
   Les clauses 1 et 2 disparaissent, il reste (¬ b ∨ b) et (¬ b ∨ c)
2. On sélectionne c qui est un littéral pur :
   La clause 2 disparait, il reste (¬ b ∨ b)
3. On sélectionne ¬ b : on supprime la clause restante

Il n'y a plus de clause, *succès* pour a=1, c=1, b=0

Dans la mise en oeuvre, on isolera correctement les fonctionnalités de recherche des mono-littéraux, des littéraux purs, des littéraux suivants, et celles de modification de l'ensemble de clauses (suppression de clauses, et suppression de littéraux dans les clauses). La liste des variables apparaissant dans l'expression de départ sera calculée une seule fois au début du traitement.

Pour que l'utilisateur puisse suivre le déroulement des deux algorithmes, on affichera à l'écran des messages informatifs sur les étapes de traitement.

On remarque que, quel que soit l'algorithme utilisé, une telle approche est exploitable pour tester si une formule F est une tautologie (i.e. est *toujours* vraie) : il suffit de tester la satisfiabilité de sa négation. En cas d'échec, F est une tautologie.

Mises en oeuvre

Le but du projet est de *comparer les deux méthodes* entre elles (avec mise sous forme normale préalable pour Davis et Putnam), et de comparer les réalisations dans les 3 langages.

Mise en oeuvre en C++ : les expressions d'origine seront codées sous forme d'arbres, on pourra réaliser une classe abstraite (d'opérateur indéterminé) et en dériver des classes filles spécifiques pour les différents opérateurs. Les clauses pourront être vues sous forme de listes, chaque littéral étant un couple (nom du littéral, signe). Ne pas hésiter à utiliser les containers de la STL.
Mise en oeuvre en Caml : les expressions d'origine seront codées sous forme d'arbres. Utiliser un (des) type(s) somme, et faire des définitions par filtrage. Les clauses seront représentées par des listes, chaque littéral étant un couple (nom du littéral, signe). Utiliser les listes et n-uplets de Caml.
Mise en oeuvre en Icon : Tout comme pour la mise en oeuvre en Caml, on utilisera des arbres, listes et articles. On utilisera les générateurs d'Icon. Une interface graphique très simple peut apporter un peu d'agrément.

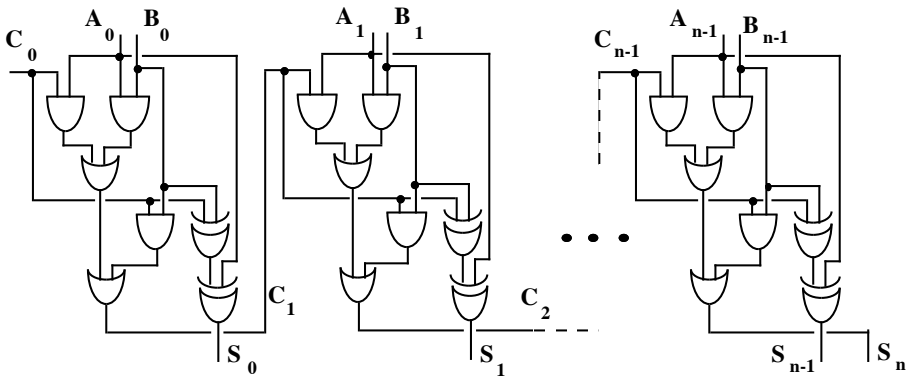
Benchmarks

1. Utilisation pour vérifier l'équivalence fonctionnelle de *deux additionneurs* combinatoires :
On utilisera les deux procédures pour vérifier l'équivalence des fonctions de sorties (il s'agit donc de tautologies) de l'additionneur à retenue anticipée 74LS283 et d'un additionneur à retenue propagée.

Additionneur à retenue anticipée :

Voir `http://www.physics.wisc.edu/graduates/courses/623-f03/ds/74LS283.pdf`

Additionneur à retenue propagée :



2. Le site `http://www.csplib.org/` propose une variété de problèmes connus pour constraint solvers et SAT solvers.
On vous demande de coder et résoudre le *problème de Schur* (voir problème numéro 15) : peut-on placer n billes numérotées de 1 à n (essayer pour différentes valeurs de n) dans m boîtes (on prendra m=3) sans qu'une seule boîte ne contienne deux billes x et y telles que x = 2y ou trois billes x, y et z telles que x + y = z ?

Enfin, il est très souhaitable que vous proposiez d'autres benchmarks !

(*) Voir par exemple :
"Intelligence artificielle et Informatique théorique", J.M.Alliot & T.Schiex, Cépaduès, 1993.