Projet 2: SAT

Julien Bensmail, Daniel Hirschkoff, Vincent Lanore

Présentation du cours

Projet2 – présentation

- implémenter plusieurs versions d'un **solveur SAT**
- dans le langage de votre choix parmi
  C Caml Java
- modalités
  - en binôme
    - appariements par "niveaux proches en programmation"
  - 5 échéances au long du semestre
- évaluation
  - capacité à développer du code
    - de façon réfléchie (clarté, modularité, efficacité)
    - de manière organisée (échéances, respect des consignes)
  - exigences
    - adaptées à votre niveau d'expertise
    - uniformité sur l'organisation
  - travail important, tout au long du semestre
- séances surtout en salle machines, quelques séances en amphi

Le problème SAT

Positionnement

- satisfiabilité: rôle central en théorie de la complexité
- des solveurs: utiliser la machine pour démontrer des résultats

  un domaine de recherche:

  des méthodes de toutes sortes. . .

  une technologie qui s'affine depuis des décennies

  . sur l'ensemble du spectre, de méthodes complètes à des heuristiques

  . de l'outil-exemple pour chercheur/théoricien à la dimension industrielle

  . . . pour résoudre maints problèmes

  logistique, planification, vérification de matériel et de logiciel, jeux, . . .
- ce cours
  - se faire de la culture sur SAT

    (est-ce vraiment un sujet de L3?) / "démystifier NP"
  - surtout :

    SAT est un support pour apprendre à programmer/s'organiser, dans le cadre d'un projet logiciel non rikiki

    (en particulier, il ne s'agit pas de faire du "SAT ultra sophistiqué")

Description, notations

- on cherche à satisfaire
  une formule logique en forme normale conjonctive

  $$(\mathtt{x}_1 \vee \overline{\mathtt{x}_3} \vee \mathtt{x}_4) \ \wedge \ (\overline{\mathtt{x}_1} \vee \mathtt{x}_2) \ \wedge \ (\mathtt{x}_2 \vee \mathtt{x}_3 \vee \mathtt{x}_5) \ \wedge \ (\overline{\mathtt{x}_3} \vee \mathtt{x}_5)$$

  quatre étages:

  - variables $x_1, x_2, \ldots, x_k$
  - littéraux α, β
    exemples: $x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots$ $\overline{x_1}$: "non x_1" (parfois $\neg x_1$)
    si $\alpha = \overline{x_7}$, α vrai signifie x_7 faux
  - clauses $C = \alpha_1 \vee \alpha_2 \vee \cdots \vee \alpha_n$
  - formule $\phi \ = \ C_1 \wedge C_2 \wedge \cdots \wedge C_r$

- formule satisfiable: il existe une *assignation* d'une valeur de vérité aux variables telle que chaque clause contienne au moins un litéral vrai
- cas limite:

  formule vide: satisfiable clause vide: insatisfiable

La méthode historique

Résolution

Résolution

- une notation pour une formule en CNF:

$$\{\{x, \overline{y}, \overline{z}\}, \{y, \overline{z}\}, \{\overline{x}\}\}$$

  $\emptyset$: satisfiable, $\{\emptyset\}$: non satisfiable

- on écrit $\phi_1 \models \phi_2$ si toute assignation satisfaisant ϕ_1 satisfait aussi ϕ_2
  - en particulier, si $C_1 \subseteq C_2$ (pour C_i des clauses), alors $\{C_1\} \models \{C_2\}$
  - si $\phi \models \{\emptyset\}$, alors ϕ est insatisfiable
- **résolution**

$$\{\{x, \alpha_1, \ldots, \alpha_k\}, \{\overline{x}, \beta_1, \ldots, \beta_n\}\} \models \{\{\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_n\}\}$$

$$\{x, \overline{x}\} \cap (\{\alpha_i\}_i \cup \{\beta_j\}_j) = \emptyset$$

  ou si on veut

$$\{C_1\}, \{C_2\} \models \underbrace{\{(C_1 \backslash \{x\}) \cup (C_2 \backslash \{\overline{x}\})\}}_{\textit{résolvant}} \quad \text{si } x \in C_1, \overline{x} \in C_2$$

Résolution – exemple, complétude

$\{\overline{x}, z\}$	(1)	$\{\overline{x}\}$	(par (1), (3))	(5)
$\{\overline{y}, z\}$	(2)	$\{\overline{y}\}$	(par (2), (3))	(6)
$\{\overline{z}\}$	(3)	$\{y\}$	(par (4), (5))	(7)
$\{x, y\}$	(4)	$\{\ \}$	(par (6), (7))	(8)

- la résolution est correcte (le résolvant découle des clauses utilisées)
- la résolution n'est pas complète (il existe des formules conséquences qui ne sont pas construites par la résolution)
- la résolution est complète *réfutationnellement*: la clause vide est obtenue si et seulement si la formule de départ est insatisfiable
- approche: appliquer la résolution jusqu'à obtenir la clause vide (insatisfiable) ou ne plus pouvoir résoudre (satisfiable)

Algorithme de Davis Putnam (1960)

- on suppose les variables ordonnées $x_1 < \cdots < x_k$
- idée: résoudre le plus possible par rapport à la plus grande variable, puis itérer

- k seaux $S_k, S_{k-1}, \ldots, S_1$
- chaque clause C est ajoutée au seau S_i si
  - x_i apparaît dans C
  - pas de variable $x_{j>i}$ dans C
- on engendre *tous les résolvants possibles* pour le seau S_k, en les insérant en-dessous dans le bon seau

  cas particulier:
  pas de résolution, car x_k n'apparaît qu'avec une seule polarité
  $\hookrightarrow$ on passe directement à S_{k-1}
- cas d'arrêt? (insatisfiable, ou solution trouvée)

Premier devoir: la résolution

le sujet sera disponible demain depuis

la page www du cours

`http://perso.ens-lyon.fr/daniel.hirschkoff/P2`

- à rendre pour le 8 février
- prochaine séance:
  - sur machines, aide au DM
  - constitution des binômes

Le sujet central de projet 2:

l'algorithme DPLL

L'algorithme au cœur du solveur: DPLL

Davis-Putnam-Logemann-Loveland, 1962

- on explore l'espace des affectations possibles des variables

  essayer avec $x_1 = \texttt{vrai}$ . . .

  essayer avec $x_2 = \texttt{vrai}\ldots$ . . . récursivement

  si ça ne marche pas,

  essayer avec $x_1 = \texttt{faux}$ . . .
- exploite ce que l'on appelle le *variable splitting*
  - on explore toutes les instanciations possibles
  - on manipule une *instanciation partielle*, que l'on étend tant qu'elle est consistante (pas de conflit)
  - en cas de conflit, on élimine un ensemble d'instanciations
- au lieu de déduire à partir des formules (sans se tromper), comme dans la résolution, on fait des paris
- à chaque pari, on **déduit des conditions nécessaires**

  p.ex. si $x_3 = \texttt{vrai}$, la clause $\overline{\texttt{x}_3} \vee \overline{\texttt{x}_{12}}$ implique $\texttt{x}_{12} = \texttt{faux}$

$\hookrightarrow$ on n'énumère ainsi pas *toutes* les affectations possibles

Algorithme DPLL — étapes essentielles

1. *boolean constraint propagation* **déduire**
   1.1 trouver les affectations *nécessaires*
      - si une clause est $\{\alpha\}$, alors α est nécessairement satisfait (i.e., $x =$ `vrai` si $\alpha = x$, $=$ `faux` si $\alpha = \overline{x}$)
      - si une variable x n'apparaît qu'avec une seule polarité, déduire sa valeur

   1.2 propagation des valeurs
   si nécessairement α est satisfait,
   . éliminer toutes les clauses contenant α
   . éliminer $\overline{\alpha}$ de toutes les clauses le contenant ($\star$)

   1.3 recommencer en 1.1
2. choix **décider**

   choisir une variable x_i encore inconnue, et lui attribuer une valeur (`vrai` ou `faux`)
3. *backtrack* **rebrousser chemin**

   l'étape ($\star$) peut engendrer une clause vide: **conflit**
   on revient alors sur le dernier *choix* ayant été fait

   (choix, pas affectation)

DPLL – exemple

autres transparents

Paris et backtrack

- à un instant donné dans l'exécution de DPLL, on a parié sur la valeur d'un certain nombre de variables
- on maintient une pile de tranches
  - on a fait n **paris** sur des variables $x_{c_1}, \ldots, x_{c_n}$
  - chaque choix a entraîné, par BCP, l'assignation d'un certain nombre de variables (elles sont dans la même tranche)
    $x_{c_1}, y^1_{c_1}, \ldots, y^{i_1}_{c_1}, \mid x_{c_2}, y^1_{c_2}, \ldots, y^{i_2}_{c_2}, \ \ldots \ \mid x_{c_n}, y^1_{c_n}, \ldots, y^{i_n}_{c_n}$
- lorsque l'on rebrousse chemin, il faut
  - retourner sa veste pour le dernier pari *où cela est possible*

    si on a parié sur $x_{c_n} = \texttt{faux}$ car on avait déjà vu que $x_{c_n} = \texttt{vrai}$ était contradictoire, remonter à $x_{c_{n-1}}$
  - annuller l'affectation des variables ayant découlé de ce dernier pari

Pensez avant de programmer

- choix des structures de données
  - qu'est-ce qu'une variable, un litéral, une clause ?
    (dans votre programme)
  - phase de propagation
    - à quoi veut-on accéder et comment?
      . toutes les clauses contenant telle variable
      . le nombre de litéraux "vivants" dans telle clause
      . le nombre de clauses "vivantes"
    - quelles opérations faut-il être en mesure d'effectuer?
  - l'affectaction courante
    que faire quand on revient sur l'affectation d'une variable?
- n'utilisez pas une grande fonction récursive: implémentez la récursion à la main (on manipule la pile, on a un meilleur contrôle)
- autre aspect important : programmez modulairement
  si dans un mois, vous voulez changer une des structures de données, il faudrait que cela ne soit pas une catastrophe
  (+ interdit de dire "désolé, mais je ne prévois pas de me tromper"!)

Rendu 1 — DPLL

- en entrée et en sortie: comme au DM
- à l'intérieur: algorithme DPLL "simple"
  - **structure modulaire**, ouvrant la voie à des raffinements successifs

    un fichier pour la boucle principale
  - **structures de données** pour
    - la représentation des variables, litéraux, clauses
      . une variable peut avoir trois états: `inconnu`, `vrai`, `faux`
    - l'état courant de la recherche
  - **efficacité raisonnable**
    - typiquement, 80-90 % du temps passé dans la propagation des informations
    - pouvoir au moins soupçonner les sources majeures d'inefficacité
  - **traitement de l'entrée**
    - propagation des unités, détecter p.ex. les $\mathtt{x} \vee \overline{\mathtt{x}}$
    - éventuellement, tri des variables
    - éventuellement, collecte d'informations sur les clauses

À titre d'information: rendus $n + 1$

- variations sur la manière dont on choisit le prochain pari
  - sur quelle variable miser, avec quelle valeur
- on adoptera une technique *astucieuse* pour la propagation des unités
- on voudra calculer des choses *astucieuses* sur l'état courant de l'exploration
- on fera du backtrack *astucieux*
  (plus en amont que le dernier pas)
- on étendra en cours de route l'ensemble des clauses
- on décidera de repartir à zéro après K conflits
- on tirera d'emblée au hasard une affectation pour toutes les variables, pour ensuite touiller de ci de là jusqu'à satisfaire la formule

ne vous effrayez pas: vous pouvez dans un premier temps faire "quelque chose qui marche", quitte à devoir reprendre une partie de la structure par la suite

Soyons alphabétisés

Lisez **les sujets de rendu.**

Divers

- `minisat` est installé sur les machines des salles libre-service
- implémentez *ce qui vous est demandé*

  notamment pour le DM
- rappel: importance de la ponctualité des rendus!
  - les rendus sont incrémentaux, mais ce n'est pas uniquement le résultat en fin de semestre qui compte

Questionnaire

puis:

- stratification des étudiants
- constitution des binômes

Watched literals

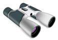
(litéraux surveillés)

Watched literals

- une technique permettant d'économiser du temps lors de la propagation des contraintes
  - *"Chaff: Engineering an Efficient SAT Solver"*, Moskewicz, Madigan, Zhao, Zhang, Malik, DAC 2001.
  - questions de droits: technique brévetée ?

- remarque:
  les clauses ont toutes au moins deux litéraux
- *propagation* lorsque tous les litéraux sauf un sont à `faux`

Watched literals — principes



- pour chaque clause, on ne regarde que deux litéraux…
  …qui ne sont pas à `faux` (`vrai` ou `inconnu`)
  - si c'est impossible, c'est que la clause en question est inactive
- lorsque le litéral α est mis à `faux`,
  - pour toutes les clauses C où α est surveillé, on cherche un autre litéral à surveiller
    - si pas possible, déclenchement de propagation supplémentaire

    NB: là où α apparaît sans être surveillé, *on ne fait rien*

  - *optionnel:* pour toutes les clauses C contenant $\overline{\alpha}$ (surveillé ou non),
    - on se fait la remarque que C est satisfaite
    - si on vient de rendre la clause satisfiable,
      on installe $\overline{\alpha}$ comme litéral surveillé (priorité aux litéraux vrais par rapport aux inconnus)

      $\hookrightarrow$ on détermine plus facilement si une clause est satisfaite

Watched literals — commentaires

- on ne regarde qu'au plus deux litéraux pour savoir si une clause est satisfaite
  invariant: tant que la clause n'est pas satisfaite, aucun des deux litéraux n'est à `faux`
- backtrack: on laisse les litéraux surveillés inchangés!

  (on ne bouge pas les jumelles)

  - l'état n'est pas nécessairement le même, mais essentiellement équivalent
  - les jumelles ne doivent pas revenir à leur position antérieure
  - remarque: si on surveille $(\texttt{faux}, \_)$, alors, par l'invariant, on surveille $(\texttt{faux}, \texttt{vrai})$, et lors d'un backtrack on passera de `faux` à `?` avant de modifier les litéraux non surveillés de la clause

- particulièrement efficace sur les entrées ayant de longues clauses

Apprentissage de clauses

(clause learning)

Étendre la formule traitée par DPLL

- principe: ajouter une **clause dérivable**
  pendant l'exécution de l'algorithme
- notations: on souhaite ajouter la possibilité de jouer un coup "apprentissage"

$$M;\, F \quad \mapsto \quad M;\, F \wedge C \qquad \text{si } F \models C$$

M: affectation courante $\quad$ F: formule traitée

  - quand construire C ?
  - comment construire C ?

Quand ça coince

- focalisons-nous sur un conflit (qui déclenche un backtrack)
  - affectation courante (litéraux décidés & litéraux déduits) **M**
  - une clause
    $$C = \alpha_1 \vee \alpha_2 \vee \cdots \vee \alpha_n$$
    telle que
    $$\mathbf{M}(\alpha_1) = \ \dots \ = \mathbf{M}(\alpha_n) = \texttt{false}$$
- d'où vient la valeur des α_i?
  - de paris faits (*paris*, ou *hypothèses*)
  - de conséquences déduites

Déroulement de l'algorithme DPLL

- niveaux de décision
  - niveau de décision: 0 au départ, $+1$ à chaque pari
  - niveau de décision d'un litéral déduit :
    niveau de décision courant au moment où la déduction est faite
- *backtrack stack*

$$|3^1 \;\; -2^1 \;\; 17^1 \;|-5^2 \;\; 4^2 \;\; -1^2 \;\; -7^2 \;|3^3 \;\; 10^3$$

(chaque litéral est annoté avec son niveau de décision)

"tranches de temps"

- transparents de Oliveras et Rodríguez-Carbonell

il y a une petite erreur : au moment de déduire p10, on se sert de la clause 4, or on ne devrait pas car c'est 2, ça n'est pas -2.

Mener l'enquête à partir d'un conflit

- on part de la clause où se manifeste le conflit

  T_0 ◇ ◇ ◇ ◇ ◇ ◇ ◇ ◇ ⟵ *tout le monde est à faux*

  $T_0 = C_i$ pour un certain i

- elle contient *au moins deux litéraux* qui ont été déterminés *(découverts faux, de fait)* au niveau de décision courant

  T_0 ◇ ◇ ◇ • ◇ ◇ • ◇

  (sinon elle aurait engendré une propagation plus tôt)

- *au moins un litéral* de la clause a été *déduit* faux au niveau de décision courant

  T_0 ◇ ◇ ◇ • ◇ ◇ • ◇

  on cherche la **justification** de cette déduction :

  c'est une clause C_j contenant l'opposé de ce litéral

  C_j ▽ $\bar{\bullet}$ ▽ ▽ ▽ • ▽ • ▽ ▽

- on construit T_1 à partir de T_0 et C_j : **résolution** via •

  T_1 ◇ ◇ ◇ ◇ ◇ • ◇ ▽ ▽ ▽ ▽ • ▽ • ▽ ▽

Engendrer de nouvelles clauses

- on “remonte la causalité” en construisant T_1 à partir de T_0

T_0 $\diamond\ \diamond\ \diamond\ \bullet\ \diamond\ \diamond\ \bullet\ \diamond$ $\qquad C_j$ $\nabla\ \overline{\bullet}\ \nabla\ \nabla\ \nabla\ \bullet\ \nabla\ \bullet\ \nabla\ \nabla$

T_1 $\diamond\ \diamond\ \diamond\ \bullet\ \diamond\ \bullet\ \diamond\ \nabla\ \nabla\ \nabla\ \nabla\ \bullet\ \nabla\ \nabla\ \nabla\ \nabla$

  - il reste au moins un $\bullet$ dans T_1, la clause résultante

    $\bullet$: litéral ayant été fixé à faux dans le niveau de décision courant

    NB: il peut n’en rester qu’un, par idempotence
  - T_1 est contradictoire, car T_0 l’est
  - T_1 est déduite à partir de $T_0 = C_i$ et C_j, donc T_1 est une conséquence du problème courant

- s’il y a au moins deux $\bullet$, on itère, en construisant T_2, T_3, ...
  - invariants :
    - il y a au moins un $\bullet$
    - T_i contradictoire
    - T_i conséquence du problème courant
  - on s’arrete lorsqu’il ne reste qu’un $\bullet$
  - ça termine : intuitivement, on ne peut pas boucler

Visualiser les justifications : Graphe des conflits

- graphe orienté, dont les nœuds sont étiquetés par des litéraux
- si lors de la propagation, on utilise la clause

$$C \;=\; \alpha_1 \vee \cdots \vee \alpha_k \vee \beta$$

pour déduire, à partir de $\overline{\alpha}_1, \ldots, \overline{\alpha}_k$, que β est vrai,

on insère les nœuds $\overline{\alpha}_1, \ldots, \overline{\alpha}_k$, et, pour chaque $\overline{\alpha}_i$,
un arc (orienté) $\overline{\alpha}_i \curvearrowright \beta$

- le conflit est représenté par un noeud supplémentaire $\perp$

$\perp$ est fils des nœuds intervenant dans le conflit

(variante: on a deux nœuds α et $\overline{\alpha}$ ayant $\perp$ comme fils)

transparents de Oliveras et Rodríguez-Carbonell (p.11)

Circonscrire le conflit

- remonter les causes du conflit nous conduit à mettre en évidence un ensemble N de nœuds
  - qui contient $\perp$
  - qui ne contient que des litéraux • dont la valeur a été *déduite* au niveau de décision courant
- la phase de "remontée" $(T_i \rightsquigarrow T_{i+1})$ s'arrête lorsque l'on trouve un UIP (*Unique Implication Point*):
  - un seul litéral du niveau courant pointe vers un nœud de N
  - tous les autres litéraux pointant vers N sont de niveau strictement inférieur au niveau courant

transparents de Oliveras et Rodríguez-Carbonell (p.12)

- on rassemble la négation des litéraux desquels part un arc vers un nœud de N, cela donne une clause conséquence

c'est la **clause apprise**

◇ ◇ ◇ ◇ ◇ ◇ ◇ ◇ •

. 1 seul litéral au niveau courant
. "explication" du conflit

Vers où backtracker?

- contemplons la clause apprise C $\diamond \diamond \diamond \diamond \diamond \diamond \diamond \diamond \bullet$

  un seul litéral, α, au niveau courant
- une telle clause
  - peut être déduite de l'ensemble des clauses manipulées par résolution $(F \models C)$
  - est contradictoire dans l'affectation courante *tous les litéraux sont à faux*
- C aurait permis de déterminer *plus tôt* la valeur de α

  p.ex., si le niveau courant est 12, avec

  $$C \quad = \quad \mathtt{x}_1^7 \vee \overline{\mathtt{x}_4}^{\,9} \vee \overline{\mathtt{x}_8}^{\,5} \vee \alpha^{\,12}$$

  on voit que la valeur de α pouvait être déterminée au niveau 9

  $\hookrightarrow$ on ajoute C et on revient au niveau 9
- backtrack : on revient au niveau

  (max des niveaux des litéraux différents de α dans C)

Backtracker loin, backtracker malin

- la clause apprise C est vue comme un *raccourci*
  - on aurait pu propager l'information plus tôt
  - et ainsi échouer plus tôt
- on remonte jusqu'à l'endroit où elle aurait pu être utilisée
  - peut-être va-t-on reparcourir le chemin qui nous a mené au conflit courant

    mêmes paris, mêmes déductions
  - on fait le pari qu'on gagne à ajouter ce raccourci

    "coupe des branches" dans l'espace des possibles
- l'UIP peut être le litéral sur lequel on a parié au niveau de décision courant

  il arrive que l'ajout de clause se ramène à revenir sur le dernier pari

Concrètement, dans votre programme

Il faut équiper les structures de données de l'information nécessaire pour expliciter les causes.

Différence par rapport à DPLL normal:

- on remplace backtrack (chronologique) par analyse de conflit
- au passage, cette version modifiée est correcte et complète, comme DPLL

Rendu 2

- pour le 22 mars   partiels?!?

- sujet “personnalisé”, par binôme