Logică și structuri discrete

Verificarea realizabilității formulelor boolene

Marius Minea

marius@cs.upt.ro

http://www.cs.upt.ro/~marius/curs/lsd/

4 noiembrie 2013

Unde *aplicăm* logica booleană ?

Cum *reprezentăm eficient* formulele boolene ?

diagrame de decizie binare

Cum *prelucrăm eficient* formulele boolene ?

verificarea realizabilității (SAT checking)

Unde aplicăm logica booleană?

Calculatoarele sunt construite din *circuite logice*
⇒ realizează aceleași *funcții* ca în logică (ȘI, SAU, NU)

Numerele sunt reprezentate în calculator *în baza 2*
⇒ valori *boolene* (F sau T, 0 sau 1)

Aritmetica pe numere e implementată prin circuite logice

```
unsigned add(unsigned a, unsigned b)
{
  // a ^ b: suma, a & b: transportul (trebuie deplasat)
  return b ? add(a ^ b, (a & b) << 1) : a; // baza: a + 0 = a
}
```

Mulțimile pot fi reprezentate prin șiruri de valori boolene
pentru fiecare element: face sau nu parte din mulțime ?

Orice noțiune din matematică sau realitate e reprezentată pe biți

O formă canonică pentru funcții boolene

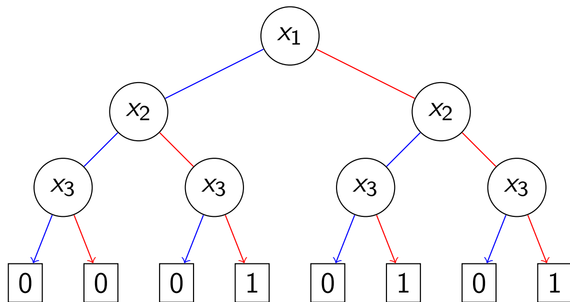
Un *circuit logic* (combinațional) e descris printr-o *funcție booleană*

Circuitul poate fi implementat (echivalent) în mai multe feluri
trebuie verificat că e într-adevăr la fel
$\Rightarrow$ că implementează *aceeași* funcție

Dar, aceeași funcție booleană se poate scrie în multe feluri!
$\Rightarrow$ vrem o reprezentare *canonică* (unică, după o anumită regulă)

Mai mult, vrem să o putem *memora* și *prelucra* eficient.

Arborele de decizie binar



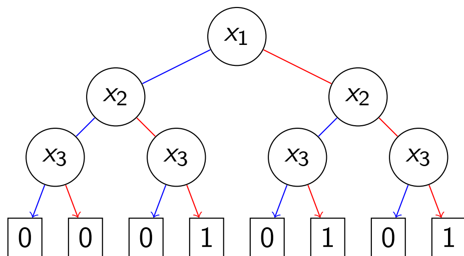
$f(x_1, x_2, x_3) = (\neg x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \neg x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_3)$

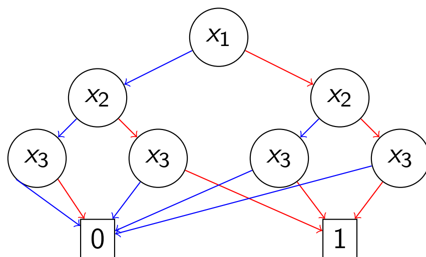
noduri *terminale*: valoarea funcției (0 sau 1)
noduri *neterminale: variabile* x_i (de care depinde funcția)
ramuri: *low*(*nod*) / *high*(*nod*) : atribuire F/T a variabilei din nod

Fixând ordinea variabilelor, arborele e unic (canonic), dar *ineficient*:
2^n combinații posibile, la fel ca tabela de adevăr

Reducerea nr. 1: Comasarea nodurilor terminale

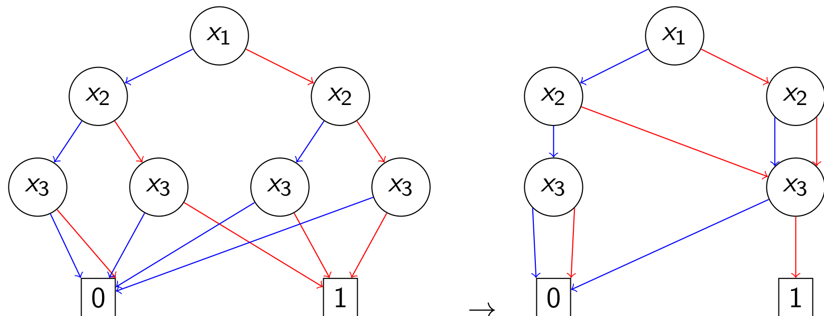
păstrăm o singură copie pentru nodurile 0 și 1





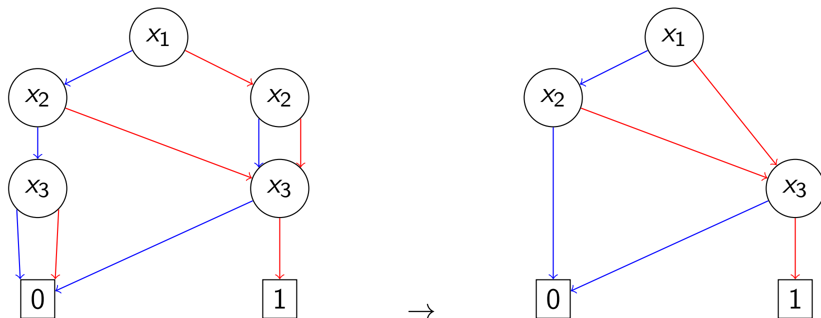
Reducerea nr. 2: Comasarea nodurilor izomorfe

Dacă $low(n_1) = low(n_2)$ și $high(n_1) = high(n_2)$, comasăm n_1 și n_2 (dacă nodurile au același rezultat pe ramura fals, și același rezultat pe ramura adevărat, ele se comportă la fel și le comasăm)



Reducerea nr. 3: Eliminarea testelor inutile

Dacă un nod dă același rezultat pe ramurile fals și adevărat, nodul poate fi eliminat



Diagrame de decizie binare

Rezultatul obținut: *binary decision diagram* (BDD)
(reprezentare introdusă de R. Bryant în 1986)

A devenit standard în industria circuitelor integrate digitale
toate companiile și programele de proiectare o folosesc

Pentru a verifica egalitatea a două funcții:
se construiesc BDD-uri pentru cele două funcții
dacă funcțiile sunt egale, se obține *același obiect* BDD
$\Rightarrow$ se verifică direct și eficient egalitatea funcțiilor

Realizabilitatea unei formule propoziționale (satisfiability)

Se dă o formulă în *logică propozițională*.
Există vreo atribuire de valori de adevăr care o face adevărată ?
= e *realizabilă* (engl. *satisfiable*) formula ?

$$\begin{aligned} & (a \vee \neg b \vee \neg d) \\ \wedge\ & (\neg a \vee \neg b) \\ \wedge\ & (\neg a \vee c \vee \neg d) \\ \wedge\ & (\neg a \vee b \vee c) \end{aligned}$$

Găsiți o atribuire care satisface formula?

Formula e în *formă normală conjunctivă* (conjunctive normal form)
= conjuncție de disjuncții de *literale* (pozitive sau negate)

Fiecare conjunct (linie de mai sus) se numește *clauză*

De ce e importantă?

Practic:

În *probleme de decizie* / constrângere:

Putem găsi o soluție la ... cu proprietatea ... ?

$\Rightarrow$ condițiile se pot exprima ca formule în logică

- în verificarea de circuite (ex. optimizăm funcția f în f_{opt})
  $f(v_1, ..., v_n) = f_{opt}(v_1, ..., v_n)$ e echivalent cu
  $f(v_1, ..., v_n) \oplus f_{opt}(v_1, ..., v_n) = 0$
  $\Rightarrow$ e corect dacă $f \oplus f_{opt}$ NU poate fi adevărată
- în verificarea de software (model checking), testare, depanare
- în biologie (determinări genetice), etc.

De ce e importantă?

Teoretic:

E prima problemă demonstrată a fi *NP-completă*.
(probleme care se crede că nu au soluții în timp polinomial)

P = clasa problemelor care pot fi rezolvate în timp polinomial
(relativ la dimensiunea problemei)

NP = clasa problemelor pentru care o soluție poate fi *verificată*
în timp polinomial (a verifica e mai ușor decât a găsi)

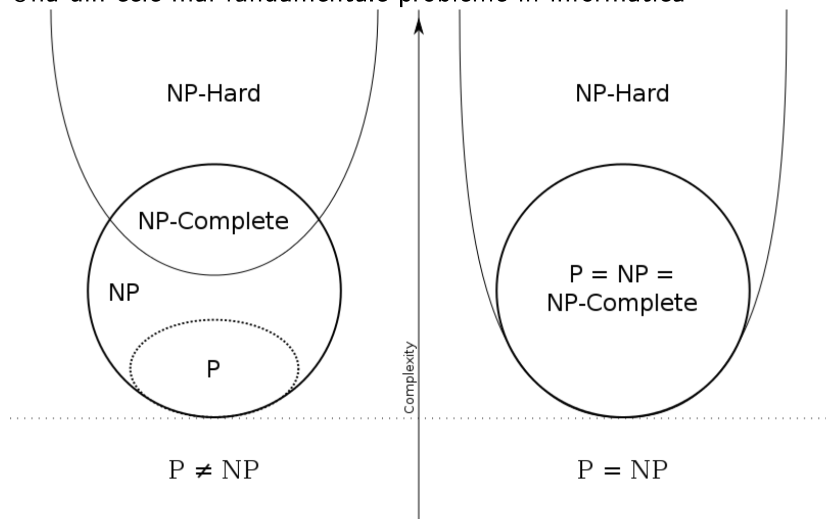
Probleme *NP-complete*: cele mai dificile probleme din clasa *NP*
dacă s-ar găsi o soluție polinomială, atunci orice problemă din NP
s-ar rezolva polinomial ⇒ P = NP

Există o colecție clasică de probleme cunoscute a fi NP-complete

Cum demonstrăm că o problemă e NP-completă (grea) ?
reducem o problemă cunoscută la problema studiată
⇒ dacă s-ar putea rezolva polinomial problema nouă,
atunci s-ar putea rezolva și problema cunoscută

P = NP?

Una din cele mai fundamentale probleme în informatică



Se crede că $P \neq NP$, dar nu s-a putut (încă) demonstra

Imagine: `http://en.wikipedia.org/wiki/File:P_np_np-complete_np-hard.svg`

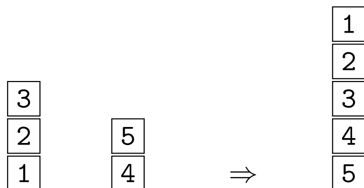
Aplicație: Planificarea

= găsirea unei secvențe de acțiuni care duc la o țintă dorită

Exemple:
deplasările unor roboți inteligenți
comportamentul sistemelor autonome (sonde spațiale)
rezolvarea de probleme (de tip puzzle, jocuri, etc.)

În general: într-un sistem descris prin *stări* și *acțiuni* (tranziții),
cum găsim o cale de la o *stare inițială* la o *stare țintă* (finală) ?

Exemplu: lumea blocurilor



Acțiuni: mutarea unui bloc liber pe alt bloc.

Ce acțiuni trebuie efectuate ? Care e numărul minim de acțiuni ?

! *Stările* și *tranzițiile* sistemului se pot reprezenta ca *formule logice*

Reprezentarea unei stări

Putem folosi *propoziții* (variabile boolene):

2		
1	3	$p_{2on1} \wedge p_{1on0} \wedge p_{3on0}$ (2 e pe 1; 1 și 3 pe masa)

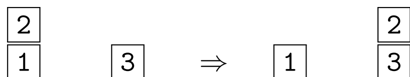
Avem nevoie de: $n \cdot (n-1)$ propoziții pentru perechi de n obiecte,
plus n propoziții care exprimă dacă un obiect e pe masă (nr. 0)

scriem și propozițiile neadevărate (din totalul de n^2 propoziții)
$\neg p_{1on2} \wedge \neg p_{1on3} \wedge \neg p_{2on0} \wedge \neg p_{2on3} \wedge \neg p_{3on1} \wedge \neg p_{3on2}$

Sau: reprezentăm *pe ce* se află fiecare piesă:
$base_1 = 0 \wedge base_2 = 1 \wedge base_3 = 0$
întregi, codificați în binar $\Rightarrow$ total $n \log n$ biți (propoziții)

Codificarea mai compactă nu duce neapărat la rezolvare eficientă.

Reprezentarea unei tranziții



Efectul mutării: p'_{2on3} (notăm cu $'$ starea următoare)

Constrângeri de execuție (starea anterioară):

$\neg p_{1on2} \land \neg p_{3on2}$ (piesa mutată e liberă)

$\neg p_{1on3} \land \neg p_{2on3}$ (piesa țintă e liberă)

Efect: $\neg p'_{2on0} \land \neg p'_{2on1}$ (2 nu va fi pe altceva)

Valorile rămân la fel pentru celelalte perechi:

$p'_{1on0} = p_{1on0} \land p'_{1on2} = p_{1on2} \land p'_{1on3} = p_{1on3}$
$\land p'_{3on0} = p_{3on0} \land p'_{3on1} = p_{3on1} \land p'_{3on2} = p_{3on2}$

Conjuncția relațiilor descrie mutarea lui 2 pe 3, în toate cazurile

Reprezentarea sistemului

Spațiul (mulțimea) *stărilor* S:
dat de propozițiile boolene $p_{i\,\mathrm{on}\,j}$, $1 \leq i \leq n, 0 \leq j \leq n$, $i \neq j$

Relația de tranziție:
se poate executa *oricare* (SAU) din tranzițiile posibile într-o stare:
p'_{1on0} (mută 1 pe masă) $\wedge$ *constrângeri mutare 1 pe 0*
$\vee$ p'_{1on2} (mută 1 pe 2) $\wedge$ *constrângeri mutare 1 pe 2*
$\vee$ ... (total 3 x 3 mutări potențiale)
$\vee$ p'_{3on2} (mută 3 pe 2) $\wedge$ *constrângeri mutare 3 pe 2*

Notăm cu $\bar{v} = \langle p_1, p_2, ..., p_N \rangle$ vectorul de stare
Relația de tranziție e o formulă $R(\bar{v}, \bar{v}')$
între starea curentă și starea următoare

! *Stările* și *tranzițiile* sistemului se reprezintă ca *formule logice*

Găsirea unui plan

Fie $S_0(\bar{v})$ și $S_f(\bar{v})$ formulele ce exprimă stările inițiale și finale
Atingerea lui S_f din S_0 în *1 mutare* = e realizabilă formula

$$S_0(\bar{v}_0) \wedge R(\bar{v}_0, \bar{v}_1) \wedge S_f(\bar{v}_1)$$

($\bar{v}_0$ e o stare inițială și $\bar{v}_1$ o stare finală și e o tranziție între ele)

Atingerea lui S_f din S_0 în *k mutări* = e realizabilă formula

$$S_0(\bar{v}_0) \wedge R(\bar{v}_0, \bar{v}_1) \wedge ... \wedge R(\bar{v}_{k-1}, \bar{v}_k) \wedge S_f(\bar{v}_k)$$

$\Rightarrow$ Găsim un plan de lungime minimă căutând succesiv soluții pentru formule tot mai complexe: $2 \cdot N$, ..., $(k+1) \cdot N$ propoziții

Există și alți algoritmi dedicați planificării.
Aici am redus problema la o exprimare *simplă*, fundamentală:
rezolvarea unei formule boolene

Cum stabilim dacă o formulă e realizabilă ?

Reguli de simplificare:

R1) Un literal *singur într-o clauză* are o singură valoare fezabilă:

în $a \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$ a trebuie să fie 1

în $(a \vee b) \wedge \neg b \wedge (\neg a \vee \neg b \vee c)$ b trebuie să fie 0

R2a) Dacă un literal e 1, *pot fi șterse clauzele* în care apare
R2b) Dacă un literal e 0, *el poate fi șters* din clauzele în care apare

Exemplele de mai sus se simplifică:

$a \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c) \quad \stackrel{a=1}{\rightarrow} \quad (b \vee c) \wedge (\neg b \vee \neg c)$

$(a \vee b) \wedge \neg b \wedge (\neg a \vee \neg b \vee c) \quad \stackrel{b=0}{\rightarrow} \quad a$

(și de aici $a = 1$, deci formula e realizabilă)

Cum stabilim dacă o formulă e realizabilă ?

R3) Dacă *nu mai sunt clauze*, am terminat (și avem o atribuire)

Dacă se ajunge la o *clauză vidă*, formula *nu e realizabilă*

$a \wedge (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$

$\stackrel{a=1}{\rightarrow} b \wedge (\neg b \vee c) \wedge (\neg b \vee \neg c)$

$\stackrel{b=1}{\rightarrow} c \wedge \neg c \quad \stackrel{c=1}{\rightarrow} \quad \emptyset$ ($\neg c$ devine clauza vidă $\Rightarrow$ nerealizabilă)

Dacă *nu mai putem face reduceri* după aceste reguli ?

$a \wedge (\neg a \vee b \vee c) \wedge (\neg b \vee \neg c) \quad \stackrel{a=1}{\rightarrow} \quad (b \vee c) \wedge (\neg b \vee \neg c)$??

R4) Alegem o variabilă și încercăm (*despărțim pe cazuri*)

- cu valoarea 1
- cu valoarea 0

O soluție pentru oricare caz e bună (nu căutăm o soluție anume).
Dacă nici un caz nu are soluție, formula nu e realizabilă.

Un algoritm de rezolvare

Problema are ca date:

- lista clauzelor (formula)
- mulțimea variabilelor deja atribuite (inițial vidă)

Regulile 1 și 2 ne *reduc problema la una mai simplă*
(mai puține necunoscute sau clauze mai puține și/sau mai simple)

Regula 3 spune când ne oprim (avem răspunsul).

Regula 4 reduce problema la rezolvarea a *două probleme mai simple*
(cu o necunoscută mai puțin)

Reducerea problemei la *aceeași problemă cu date mai simple*
(una sau mai multe instanțe) înseamnă că problema e *recursivă*.

Obligatoriu: trebuie să avem și o *condiție de oprire*

Algoritmul Davis-Putnam-Logemann-Loveland

```
function solve(env: lit set, clauses: lit list list)
(newenv, clauses) = simplify(env, clauses) (* R1, R2 *)
if clauses = empty list then
   return  env; (* atribuirile la propozitii *)
if clauses has empty clause then
   return  false; (* nerealizabila *)
if clauses contains single literal a then
   solve (env with a=true, clauses)
else
   return  solve (env with a=false, clauses)
        or solve (env with a=true, clauses);
```

Cu optimizări poate rezolva formule cu > 1 milion de variabile

Implementare: lucrul cu liste și mulțimi

Structuri de date:

- *lista* clauzelor (listă de liste de literale)
- *mulțimea* literalelor cu valoare 1 (T)

Prelucrări:

- *căutarea* unui literal în mulțimea celor atribuite
- *adăugarea* unui literal la mulțimea celor atribuite
- *parcurgerea* literalelor dintr-o listă (clauză)
- *eliminarea* unui literal dintr-o listă (clauză)
- *eliminarea* unei clauze dintr-o listă (formula)

Cum reprezentăm un literal ?

Vrem cod independent de reprezentarea literalelor (șiruri, întregi...)
Trebuie: să putem nega un literal, și să putem crea mulțimi.
Definim *semnătura* (interfața) unui tip și un modul de implementare

```
module type LITERAL = sig            (* interfata *)
  type t
  val compare: t -> t -> int (* necesar pt. multimi *)
  val neg: t -> t
end
module StrLit = struct (* instantiem tipul propriu-zis *)
  type t = P of string | N of string (* pozitiv / negat *)
  let compare = Pervasives.compare (* functie standard *)
  let neg = function
    | P s -> N s
    | N s -> P s
end
```

(cod după Conchon et. al, SAT-MICRO, 2008)

Un modul parametrizat

Creăm un modul care poate lucra cu orice literal care satisface interfața (semnătura) `LITERAL` definită.
$\Rightarrow$ putem schimba oricând reprezentarea, păstrând codul

```
module Sat(L: LITERAL) = struct

  module S = Set.Make(L) (* tipul multime de literale *)

  exception Unsat (* exceptie daca nu e realizabila *)

  (* aici definim functiile modulului *)
end
```

Simplificarea unei clauze

Păstrăm doar literalele care *nu* apar negate în `ones` (R2b)
(mulțimea literalelor adevărate)

```
let rec filter_clause ones = List.filter (fun e ->
  if S.mem e ones then raise Exit
  else not (S.mem (L.neg e) ones))
```

Găsirea unui literal adevărat e un caz special (R2a)
⇒ nu mai continuăm prelucrarea clauzei (*excepția* `Exit`)
Altfel, păstrăm doar literalele care nu sunt sigur false
(nu apar negați în mulțimea celor adevărate, `ones`)

Simplificarea listei de clauze

Acumulăm cu `List.fold_left` atât mulțimea de literale *true* cât și clauzele modificate:

```
let rec simplify ones = List.fold_left
  (fun (ones, clst) cl ->
    try (match filter_clause ones cl with
    | [] -> raise Unsat (* clauza vida -> nerealizabila *)
    | [lit] -> simplify (S.add lit ones) clst (* reia cu lit=T
    | newcl -> (ones, newcl :: clst))
    with Exit -> (ones, clst) (* clauza T -> nu modifica *)
  ) (ones, [])
```

Dacă `filter_clause` dă un singur literal, reluăm simplificarea, adăugându-l la cele adevărate
Dacă returnează lista vidă, toată formula e nerealizabilă
Dacă iese cu excepția `Exit`, ignorăm clauza (e adevărată)
Altfel, adăugăm clauza simplificată la listă

Verificarea propriu-zisă

Dacă după simplificare obținem lista vidă de clauze, returnăm lista literalelor adevărate (restul nu contează)

Altfel, cu primul literal din prima clauză încercăm ambele valori
  dacă prima încercare dă excepția `Unsat`, încercăm și a doua

```
let sat =
  let rec sat1 ones1 clst =
  match simplify ones1 clst with
  | (ones, []) -> S.elements ones
  | (ones, (lit::cls)::clist) ->
    try sat1 (S.add (L.neg lit) ones) (cls::clist)
    with Unsat -> sat1 (S.add lit ones) clist
  in sat1 S.empty        (* initial fara atribuiri *)
```

În final, folosim:

```
open StrLit (* pentru a putea folosi P, N *)
module SatS = Sat(StrLit);; (* instantiem modulul *)
SatS.sat[[P "a"; P "b"; N "c"]; [N "a"; P "c"]; [P "a"; N "b"]]
- : SatS.S.elt list = [N "a"; N "b"; N "c"]
```