Logică și structuri discrete

Logică propozițională

Marius Minea
marius@cs.upt.ro

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

3 noiembrie 2015

Logica stă la baza informaticii

circuite logice: descrise în algebra booleană

calculabilitate: ce se poate calcula algoritmic?

metode formale: demonstrarea corectitudinii programelor

inteligența artificială: cum reprezentăm și deducem cunoștințe?

baze de date relaționale

etc.

Din istoria logicii

Aristotel (sec.4 î.e.n.): primul sistem de *logică formală* (riguroasă)

Gottfried Wilhelm Leibniz (1646-1714): *logică computațională*
 raționamentele logice pot fi reduse la *calcul matematic*

George Boole (1815-1864): *The Laws of Thought*:
 logica modernă, algebră booleană (logică și mulțimi)

Gottlob Frege (1848-1925): *logica simbolică clasică*
 Begriffsschift: formalizare a logicii ca fundament al matematicii

Bertrand Russell (1872-1970): *Principia Mathematica*
(cu A. N. Whitehead)
 formalizare încercând să elimine paradoxurile anterioare

Kurt Gödel (1906-1978): *teoremele de incompletitudine* (1931):
 nu există axiomatizare consistentă și completă a aritmeticii

Operatori logici uzuali

Știm deja: operatorii logici NU ($\neg$), SAU ($\vee$), ȘI ($\wedge$)

Tabele de adevăr:

p	$\neg p$
F	T
T	F

negație $\neg$ NU
C: `!` ML: `not`

	$p \vee q$	q: F	q: T
p	F	F	T
p	T	T	T

disjuncție $\vee$ SAU
C/ML: `||`

	$p \wedge q$	q: F	q: T
p	F	F	F
p	T	F	T

conjuncție $\wedge$ ȘI
C/ML: `&&`

Logica propozițională

Unul din cele mai simple *limbaje* (limbaj $\Rightarrow$ putem *exprima* ceva)
așa cum codificăm numere, etc. în *biți*
putem exprima probleme prin *formule* în logică

Discutăm:

Cum definim o *formulă logică*:
forma ei (*sintaxa*) vs. înțelesul ei (*semantica*)

Cum *reprezentăm* o formulă? pentru a opera *eficient* cu ea

Cum *folosim* logica pentru a lucra cu alte noțiuni din informatică?
(mulțimi, relații, automate)

Ce sunt *demonstrațiile* și *raționamentul logic* ?
cum putem demonstra? se poate demonstra (sau nega) orice?

Propoziții logice

O *propoziție* (logică) e o afirmație care e fie adevărată, fie falsă, dar nu ambele simultan.

Sunt sau nu propoziții?

$2 + 2 = 5$

$\mathrm{x} + 2 = 4$

Toate numerele prime mai mari ca 2 sunt impare.

$x^n + y^n = z^n$ nu are soluții întregi nenule pentru niciun $n > 2$

Dacă $x < 2$, atunci $x^2 < 4$

Sintaxa logicii propoziționale

Un *limbaj* e definit prin *simbolurile* sale și prin *regulile* după care le combinăm corect.

Simbolurile logicii propoziționale:

propoziții: notate deobicei cu litere p, q, r, etc.

operatori (conectori logici): negație $\neg$, implicație $\rightarrow$, paranteze ( )

Formulele logicii propoziționale: definite prin *inducție structurală*, o formulă complexă e construită din formule mai simple

orice *propoziție* (numită și formulă atomică)

$(\neg\alpha)$ dacă α este o formulă

$(\alpha \rightarrow \beta)$ dacă α și β sunt formule

(α, β numite *subformule*)

Omitem parantezele redundante, considerând $\neg$ mai prioritar ca $\rightarrow$

Operatorii cunoscuți pot fi definiți folosind $\neg$ și $\rightarrow$:

$\alpha \wedge \beta \stackrel{def}{=} \neg(\alpha \rightarrow \neg\beta)$ (ȘI) $\alpha \vee \beta \stackrel{def}{=} \neg\alpha \rightarrow \beta$ (SAU)

$\alpha \leftrightarrow \beta \stackrel{def}{=} (\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$ (echivalență)

Sintaxa (concretă și abstractă) vs. semantică

Sintaxa: o mulțime de reguli care definește construcțiile unui limbaj
aici: *formulele* logicii propoziționale
NU spune *ce înseamnă* o formulă

Semantica: definește înțelesul unei construcții (unui limbaj)

Sintaxa *concretă* precizează modul *exact* de scriere. O formulă e:
prop $\neg$ *formulă* *formulă* $\wedge$ *formulă* sau *formulă* $\vee$ *formulă*

Sintaxa *abstractă*: interesează *structura* formulei din subformule:
propoziție, negația unei formule, conjuncția/disjuncția a 2 formule

În ML, putem definim un tip recursiv urmărind *sintaxa abstractă*:

```
type boolform = V of string | Neg of boolform
  | And of boolform * boolform | Or of boolform * boolform
```

Numele de *constructori* `And`, `Or` sunt alese de noi
Tipul reprezintă *structura* formulelor, nu simboluri concrete ($\wedge$, $\vee$),
nici scrierea infix sau prefix, etc.

Implicația logică $\rightarrow$

$p \rightarrow q$ numită și *condițional(ă)*
p: *antecedent* (sau *ipoteză*)
q: *consecvent* (sau *concluzie*)

Semnificație: dacă p e adevărat, atunci q e adevărat (if-then)
dacă p nu e adevărat, nu știm nimic despre q (poate fi oricum)

Deci, $p \rightarrow q$ e fals doar dacă p e adevărat și q e fals
(dacă p, atunci q ar trebui să fie adevărat)

Tabelul de adevăr:

$$\begin{array}{cc|cc} & & \multicolumn{2}{c}{q} \\ & p \rightarrow q & \text{F} & \text{T} \\ \hline p & \text{F} & \text{T} & \text{T} \\ \hline & \text{T} & \text{F} & \text{T} \end{array}$$

Exprimat cu alți conectori: $\boxed{p \rightarrow q \; = \; \neg p \vee q}$

Implicația în vorbirea curentă și în logică

În limbajul natural, “dacă ... atunci” denotă deobicei *cauzalitate*
dacă plouă, iau umbrela

În logica matematică, $\rightarrow$ *NU înseamnă cauzalitate*
3 e impar $\rightarrow$ 2 e număr prim   implicație adevărată, T $\rightarrow$ T

În demonstrații, vom folosi ipoteze *relevante* (legate de concluzie)

Vorbind, spunem adesea “dacă” gândind “dacă și numai dacă”
(echivalență, o noțiune mai puternică!)
Exemplu: Dacă depășesc viteza, iau amendă.

ATENȚIE: *fals implică orice!* (vezi tabelul de adevăr)
$\Rightarrow$ un raționament cu o verigă falsă poate duce la orice concluzie
$\Rightarrow$ un paradox ($p \wedge \neg p$) distruge încrederea într-un sistem logic

Despre implicație

Fiind dată o implicație $p \to q$, definim:

reciproca: $q \to p$

inversa: $\neg p \to \neg q$

contrapozitiva: $\neg q \to \neg p$

Contrapozitiva e *echivalentă* cu formula inițială (directa).
Inversa e echivalentă cu reciproca.

$p \to q$ ~~NU e echivalent~~ cu $q \to p$ (reciproca)

Calculul în logică: funcții de adevăr

Formula e o noțiune *sintactică*.
Valoarea de adevăr e o noțiune *semantică*.

Definim riguros cum calculăm valoarea de adevăr a unei formule.

O *funcție de adevăr* v atribuie la orice formulă o *valoare de adevăr* $\{\mathsf{T}, \mathsf{F}\}$ astfel încât:

- $v(p)$ e definită pentru fiecare *propoziție* atomică p.
- $$v(\neg\alpha) = \begin{cases} \mathsf{T} & \text{dacă } v(\alpha) = \mathsf{F} \\ \mathsf{F} & \text{dacă } v(\alpha) = \mathsf{T} \end{cases}$$
- $$v(\alpha \to \beta) = \begin{cases} \mathsf{F} & \text{dacă } v(\alpha) = \mathsf{T} \text{ și } v(\beta) = \mathsf{F} \\ \mathsf{T} & \text{în caz contrar} \end{cases}$$

Exemplu: fie $v(a) = \mathsf{T}$, $v(b) = \mathsf{F}$, $v(c) = \mathsf{T}$
putem calcula v pentru orice formulă cu propoziții din $\{a, b, c\}$

$v((a \to b) \to c)$:
avem $v(a \to b) = \mathsf{F}$ pentru că $v(a) = \mathsf{T}$ și $v(b) = \mathsf{F}$
și atunci $v((a \to b) \to c) = \mathsf{T}$ pentru că $v(a \to b) = \mathsf{F}$.

Interpretări ale unei formule

O *interpretare* a unei formule = o evaluare pentru propozițiile ei

O intrepretare *satisface* o formulă dacă o evaluează la T.
Spunem că interpretarea e un *model* pentru formula respectivă.

Exemplu: pentru formula $a \wedge (\neg b \vee \neg c) \wedge (\neg a \vee c)$
interpretarea $v(a) = \mathrm{T}, v(b) = \mathrm{F}, v(c) = \mathrm{T}$ o satisface
interpretarea $v(a) = \mathrm{T}, v(b) = \mathrm{T}, v(c) = \mathrm{T}$ nu o satisface.

O formulă poate fi:
tautologie (*validă*): adevărată în *toate* interpretările
realizabilă (en. *satisfiable*): adevărată în *cel puțin o* interpretare
contradicție (nerealizabilă): nu e adevărată în *nicio* interpretare
contingentă: adevărată în unele interpretări, falsă în altele
(nici tautologie, nici contradicție)

Tabela de adevăr

Tabela de adevăr prezintă valoarea de adevăr a unei formule în *toate interpretările posibile*

2^n interpretări dacă formula are n propoziții

Două formule sunt *echivalente* dacă au *același tabel de adevăr*

Două formula ϕ și ψ sunt echivalente dacă $\phi \leftrightarrow \psi$ e o tautologie

Exemple de tautologii

$a \vee \neg a$

$\neg\neg a \leftrightarrow a$

$$\neg(a \vee b) \;\leftrightarrow\; \neg a \wedge \neg b$$

$$\neg(a \wedge b) \;\leftrightarrow\; \neg a \vee \neg b$$

(regulile lui de Morgan)

$$(a \to b) \wedge (\neg a \to c) \;\leftrightarrow\; (a \wedge b) \vee (\neg a \wedge c)$$

$$a \to (b \to c) \;\leftrightarrow\; (a \wedge b) \to c$$

$$(p \to q) \wedge p \;\to\; q$$

$$(p \to q) \wedge \neg q \;\to\; \neg p$$

$$p \wedge q \;\to\; p$$

$$(p \to q) \;\to\; q$$

$$(p \vee q) \wedge \neg p \;\to\; q$$

$$(p \to q) \wedge (q \to r) \;\to\; (p \to r)$$

Algebra Booleană

Pe mulțimi, $\cup$, $\cap$ și complementul formează o algebră booleană.
Tot o algebră booleană formează în logică și $\wedge$, $\vee$ și $\neg$:

Comutativitate: $A \vee B = B \vee A \qquad A \wedge B = B \wedge A$

Asociativitate: $(A \vee B) \vee C = A \vee (B \vee C)$ și
$(A \wedge B) \wedge C = A \wedge (B \wedge C)$

Distributivitate: $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$ și
$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$

Identitate: există două valori (aici F și T) astfel ca:
$$A \vee \mathrm{F} = A \qquad A \wedge \mathrm{T} = A$$

Complement: $A \vee \neg A = \mathrm{T} \qquad A \wedge \neg A = \mathrm{F}$

Alte proprietăți (pot fi deduse din cele de mai sus):

Idempotență: $A \wedge A = A \qquad A \vee A = A$

Absorbție: $A \vee (A \wedge B) = A \qquad A \wedge (A \vee B) = A$
$\neg A \vee (A \wedge B) = \neg A \vee B$ (din distributivitate și complement)

Forma normală conjunctivă (conjunctive normal form)

formula = *conjuncție* $\wedge$ de *clauze*
clauză = *disjuncție* $\vee$ de *literale*
literal = propoziție sau negația ei

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

Similar: forma normală *disjunctivă* (disjuncție de conjuncții)

Transformarea unei formule în formă normală conjunctivă

ducem (repetat) negația înăuntru (regulile lui de Morgan)

$$\neg(A \vee B) = \neg A \wedge \neg B \qquad \neg(A \wedge B) = \neg A \vee \neg B$$

ducem (repetat) disjuncția înăuntru (distributivitate)

$$(A \wedge B) \vee C = (A \vee C) \wedge (B \vee C)$$

Abordarea naivă poate *crește exponențial* dimensiunea formulei:
$(p_1 \wedge p_2 \wedge p_3) \vee (q_1 \wedge q_2 \wedge q_3) =$
$(p_1 \vee (q_1 \wedge q_2 \wedge q_3)) \wedge (p_2 \vee (q_1 \wedge q_2 \wedge q_3)) \wedge (p_3 \vee (q_1 \wedge q_2 \wedge q_3))$
$= (p_1 \vee q_1) \wedge (p_1 \vee q_2) \wedge (p_1 \vee q_3) \wedge (p_2 \vee q_1) \wedge (p_2 \vee q_2) \wedge (p_2 \vee q_3)$
$\wedge (p_3 \vee q_1) \wedge (p_3 \vee q_2) \wedge (p_3 \vee q_3)$

Transformarea Tseitin (1968)

Dă o formulă realizabilă *dacă și numai dacă* cea inițială e realizabilă
echirealizabilă (en. *equisatisfiable*)

Dimensiunea formulei rezultante e *liniară* în cea a formulei inițiale

Pentru fiecare operator introducem *o nouă propoziție*
reprezintă subformula calculată de acel operator

Scriem (în CNF) că noua propoziție e *echivalentă* cu subformula
(implicație în ambele sensuri)

Obținem regulile de transformare pentru cei trei operatori

formula	$p \leftrightarrow$ formula	rescriere in CNF
$\neg A$	$(\neg A \rightarrow p) \wedge (p \rightarrow \neg A)$	$(A \vee p) \wedge (\neg A \vee \neg p)$
$A \wedge B$	$(A \wedge B \rightarrow p) \wedge (p \rightarrow A \wedge B)$	$(\neg A \vee \neg B \vee p) \wedge (A \vee \neg p) \wedge (B \vee \neg p)$
$A \vee B$	$(p \rightarrow A \vee B) \wedge (A \vee B \rightarrow p)$	$(A \vee B \vee \neg p) \wedge (\neg A \vee p) \wedge (\neg B \vee p)$

Rezultă o formulă cu mai multe propoziții $\Rightarrow$ *nu e echivalentă*
dar e realizabilă dacă și numai dacă formula inițială e realizabilă
deci o putem folosi în verificarea realizabilității

Transformarea Tseitin: exemplu

Numerotăm fiecare operator din formulă: $(a \overset{1}{\wedge} \neg b) \vee \overset{2}{\neg} (c \overset{3}{\wedge} d)$
nu numerotăm negațiile aplicate direct la propoziție $\neg b$
și nici conectorii la nivelul cel mai de sus $(...) \vee (...)$

Introducem propozițiile: $p_1 \leftrightarrow a \overset{1}{\wedge} \neg b$, $p_2 \leftrightarrow \overset{2}{\neg} p_3$, $p_3 \leftrightarrow c \overset{3}{\wedge} d$.

Scriem relațiile intrare-ieșire pentru fiecare operator în parte
și adăugăm formula pentru operatorul care dă rezultatul.
Legăm toate acest relații prin conjuncție:

$(\neg a \vee b \vee p_1) \wedge (a \vee \neg p_1) \wedge (\neg b \vee \neg p_1)$ $\quad$ p_1 reprezintă $a \wedge \neg b$

$\wedge\, (p_3 \vee p_2) \wedge (\neg p_3 \vee \neg p_2)$ $\quad$ p_2 reprezintă $\neg p_3$

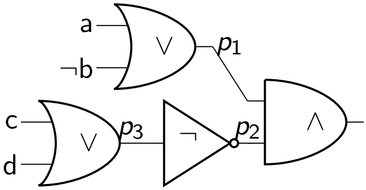
$\wedge\, (\neg c \vee \neg d \vee p_3) \wedge (c \vee \neg p_3) \wedge (d \vee \neg p_3)$ $\quad$ p_3 reprezintă $c \wedge d$

$\wedge\, (p_1 \vee p_2)$ $\quad$ vrem ca toată formula să fie adevărată

Putem transforma direct conjuncții/disjuncții multiple
ȘI $\quad (\neg A_1 \vee \neg A_2 \vee \neg A_3 \vee p) \wedge (A_1 \vee \neg p) \wedge (A_2 \vee \neg p) \wedge (A_3 \vee \neg p)$
ȘAU $\quad (A_1 \vee A_2 \vee A_3 \vee \neg p) \wedge (\neg A_1 \vee p) \wedge (\neg A_2 \vee p) \wedge (\neg A_3 \vee p)$

Transformarea Tseitin: formula ca circuit logic

$(a \overset{1}{\vee} \neg b) \wedge \overset{2}{\neg} (c \overset{3}{\vee} d)$
intrarea fiecărei porți: propoziție nouă
nu pentru propozitiile inițiale
și nici negațiile lor
Exprimăm fiecare operator:
$(p_1 \leftrightarrow a \vee \neg b) \wedge (p_3 \leftrightarrow c \vee d)$
$\wedge (p_2 \leftrightarrow \neg p_3) \wedge p_1 \wedge p_2$
ultimul termen dă rezultatul: $p_1 \wedge p_2$



Scriem fiecare echivalență în CNF, sau direct după regulile dinainte; punem conjuncție între ele

$(a \vee \neg b \vee \neg p_1) \wedge (\neg a \vee p_1) \wedge (b \vee p_1)$
$\wedge (p_3 \vee p_2) \wedge (\neg p_3 \vee \neg p_2)$
$\wedge (c \vee d \vee \neg p_3) \wedge (\neg c \vee p_3) \wedge (\neg d \vee p_3)$
$\wedge p_1 \wedge p_2$ (nivelul final cu rezultatul)

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*

Unde se aplică determinarea realizabilității?

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})
  dacă $f(v_1, ..., v_n) \leftrightarrow f_{opt}(v_1, ..., v_n)$ (echivalente)
  atunci $\neg(f(v_1, ..., v_n) \leftrightarrow f_{opt}(v_1, ..., v_n))$ e nerealizabilă
  putem verifica dacă transformarea (optimizarea) e corectă
- în verificarea de software (model checking), testare, depanare
  determinarea de teste care duc programul pe o anume cale
  găsirea de vulnerabilități de securitate în software
- în biologie (determinări genetice), etc.

Complexitatea realizabilității

n propoziții: 2^n atribuiri $\Rightarrow$ timp *exponențial* încercând toate
O atribuire dată se verifică în timp *liniar* (în dimensiunea formulei)

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)

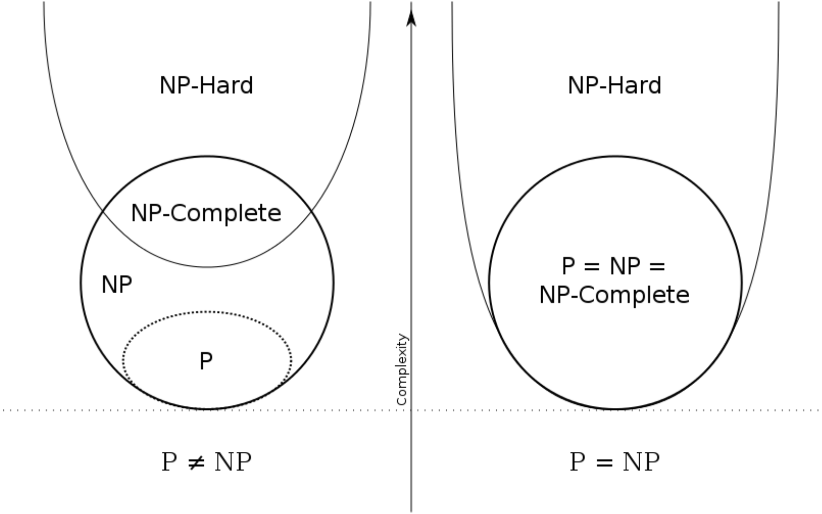
Probleme *NP-complete*: cele mai dificile probleme din clasa *NP*
dacă s-ar rezolva în timp polinomial, orice altă problemă din NP s-ar rezolva în timp polinomial $\Rightarrow$ ar fi P = NP (se crede P $\neq$ NP)

Realizabilitatea (SAT) e prima problemă demonstrată a fi *NP-completă* (Cook, 1971). Sunt multe altele (21 probleme clasice: Karp 1972).

Cum demonstrăm că o problemă e NP-completă (grea) ?
reducem o problemă cunoscută din NP la problema studiată
$\Rightarrow$ dacă s-ar putea rezolva în timp polinomial problema nouă, atunci ar lua timp polinomial 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`

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 T

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

R2a) Dacă un literal e T, *pot fi șterse clauzele* în care apare
(ele sunt adevărate, și nu mai influențează formula)

R2b) Dacă un literal e F, *el poate fi șters* din clauzele în care apare
(nu ajută în a face clauza adevărată)

Exemplele de mai sus se simplifică:

$$a \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c) \quad \overset{a=\mathrm{T}}{\rightarrow} \quad (b \vee c) \wedge (\neg b \vee \neg c)$$

$$(a \vee b) \wedge \neg b \wedge (\neg a \vee \neg b \vee c) \quad \overset{b=\mathrm{F}}{\rightarrow} \quad a$$

(și de aici $a = T$, 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*

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

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

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

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

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

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

- cu valoarea F
- cu valoarea T

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 (1962)

function solve(truelit: lit set, clauses: lit list list)
(truelit, clauses) = simplify(truelit, clauses) (* R1, R2 *)
if clauses = lista vidă **then**
  return truelit; (* R3: realizabila, returneaza atribuirile *)
if clauses conține clauza vidă **then**
  raise Unsat; (* R3: nerealizabila *)
if clauses conține clauză cu unic literal *a* **then**
  solve (truelit $\cup\{a\}$, clauses) (* R1: *a* trebuie să fie T *)
else
  try solve (truelit $\cup\{\neg a\}$, clauses); (* R4: încearcă a=F *)
  with Unsat $\rightarrow$ solve (truelit $\cup\{a\}$, clauses); (* încearcă T *)

Rezolvitoarele (*SAT solvers/checkers*) moderne pot rezolva formule cu milioane de variabile (folosind optimizări)

Implementare: lucrul cu liste și mulțimi

Structuri de date:

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

un șir (numele variabilei) etichetat cu P (pozitiv) / N (negativ)

```
module L = struct
  type t = P of string | N of string (* pozitiv / negat *)
  let compare = compare   (* fct. std. Pervasives.compare *)
  let neg = function      (* negare = schimba eticheta *)
    | P s -> N s
    | N s -> P s
 end
 module S = Set.Make(L)  (* pentru multimi de literale *)
```

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

Simplificarea unei clauze

`ones` = mulțimea literalelor adevărate

Găsirea unui literal adevărat e un caz special (R2a)
$\Rightarrow$ nu mai continuăm prelucrarea clauzei (*excepția* `Exit`)

Altfel, păstrăm doar literalele care nu sunt sigur false (R2b)
(nu apar negate în mulțimea celor adevărate, `ones`)

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

Simplificarea listei de clauze

Acumulăm cu `List.fold_left` o *pereche* de valori:
mulțimea de literale adevărate și lista clauzelor modificate

```
let rec simplify ones = List.fold_left
  (fun (ones, clst) cl ->      (* cl = clauza curenta *)
    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 unic literal, se adaugă la cele adevărate
și reluăm simplificarea clauzelor deja prelucrate
Dacă returnează lista vidă, toată formula e nerealizabilă
Dacă produce excepția `Exit`, clauza nu are efect (e adevărată)
Altfel, adăugăm clauza simplificată la listă

Verificarea propriu-zisă

Dacă simplificând obținem lista vidă de clauze, returnăm mulțimea 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 clst =
  let rec sat1 ones clst =
    match simplify ones clst with
    | (ones, []) -> ones          (* literale adevarate *)
    | (ones, (lit::cl)::clst) ->  (* luam primul literal *)
      S.union ones (     (* cei deja true + nou aflati *)
        try sat1 (S.singleton (L.neg lit)) (cl::clst) (* lit=F *)
        with Unsat -> sat1 (S.singleton lit) clst     (* lit=T *)
      )
  in S.elements (sat1 S.empty clst) (* pornim fara atribuiri *)

sat [[P "a"; P "b"; N "c"]; [N "a"; P "c"]; [P "a"; N "b"]]
- : S.elt list = [N "a"; N "b"; N "c"]
```

$(a \vee b \vee \neg c) \wedge (\neg a \vee c) \vee (\neg a \vee b)$ e realizabilă cu $a = b = c = \mathrm{F}$