Logică și structuri discrete

Complexitate și Calculabilitate

Marius Minea
marius@cs.upt.ro

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

9 ianuarie 2017

Câteva întrebări practice în calculatoare

Se poate scrie un antivirus perfect ?
(detectează toți virușii, nimic altceva)

Se poate scrie compilatorul care optimizează cel mai bine ?

Se poate crea o inteligență artificială care să producă
alta și mai inteligentă ?

Complexitatea algoritmică

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)

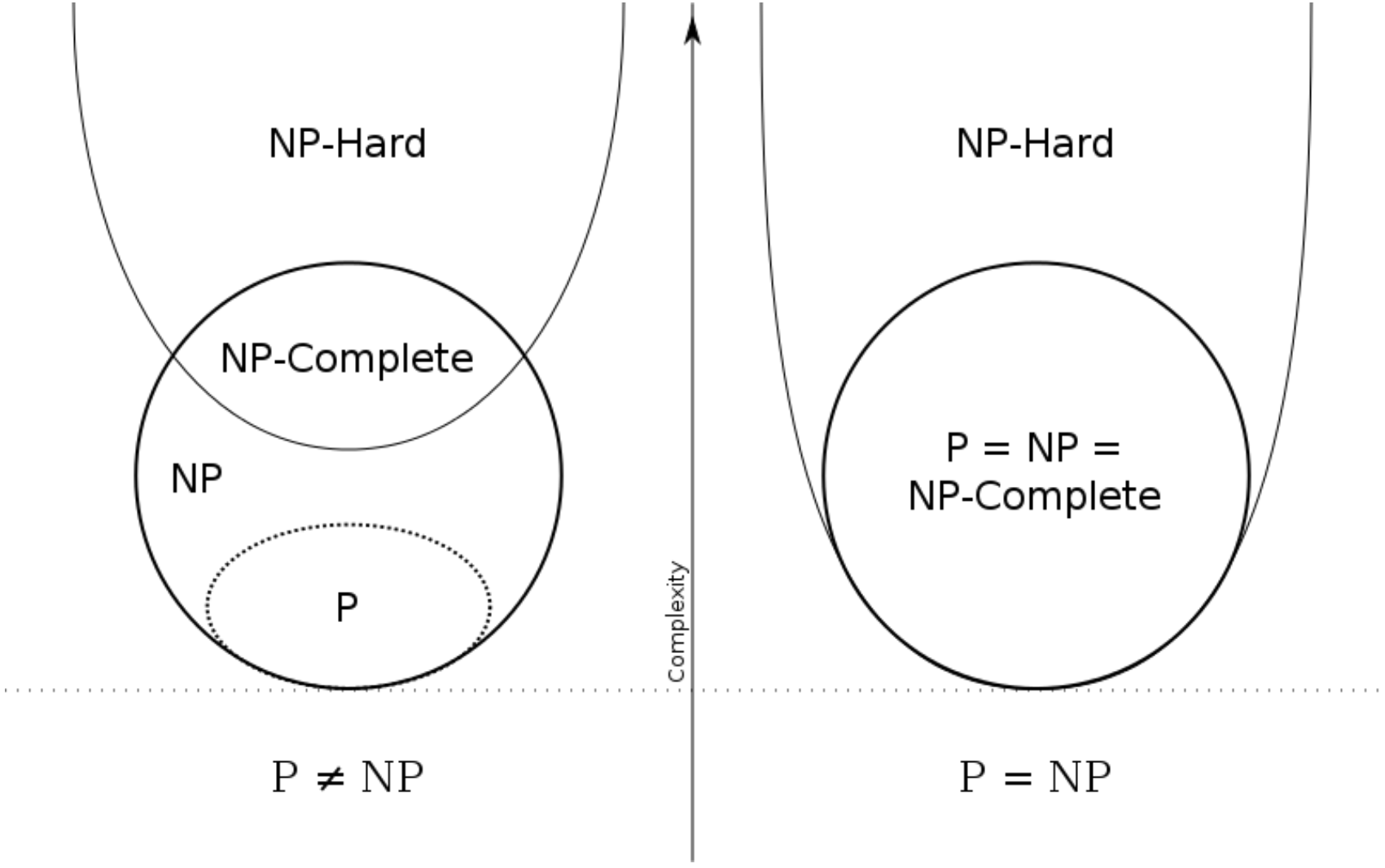
Probleme *NP-complete*: cele mai dificile probleme din clasa *NP*
dacă s-ar rezolva în timp polinomial, orice altă problemă din NP
s-ar rezolva în timp polinomial $\Rightarrow$ ar fi P = NP (se crede P $\neq$ NP)

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

Demonstrație (sintactică) și consecință logică (semantică)

Spre deosebire de logica propozițională, în logica predicatelor, numărul interpretărilor e *infinit*

$\Rightarrow$ nu mai putem construi exhaustiv tabelul de adevăr.

E *esențial* deci să putem *demonstra* o formulă
(pornind de la axiome și regiuli de inferență)

Deducția (demonstrația) se face pur sintactic.
Consecința/implicația logică e o noțiune semantică, considerând *interpretări* și valori de adevăr.

Implicația logică (consecința semantică):
Fie H o mulțime de formule și φ o formulă.
Spunem că H implică φ ($H \models \varphi$) dacă pentru orice interpretare I,

$$I \models H \text{ implică } I \models \varphi$$

(φ e adev. în orice interpretare care satisface toate ipotezele din H)

Consistență și completitudine

Calculul predicatelor de ordinul I este *consistent* și *complet* (la fel ca și logica propozițională):

$$H \vdash \varphi \text{ dacă și numai dacă } H \models \varphi$$

Dar: relația de implicație logică e doar *semidecidabilă*

dacă o formulă e o tautologie, ea poate fi demonstrată

dar dacă nu e, încercarea de a o demonstra (sau o refuta) poate continua la nesfârșit

Teoremele de incompletitudine ale lui Gödel

Prima teoremă de incompletitudine:
Orice sistem logic capabil să exprime aritmetica elementară nu poate fi și consistent și complet

i.e., se poate scrie o afirmație adevărată dar care nu poate fi demonstrată în acel sistem

Demonstrație: codificând formule și demonstrații ca numere construim un număr care exprimă că formula sa e nedemonstrabilă

A doua teoremă de incompletitudine:
Consistența unui sistem logic capabil să exprime aritmetica elementară nu poate fi demonstrată în cadrul acelui sistem.

dar ar putea fi demonstrabilă în alt sistem logic (mai bogat)

Logică și calcul (programare)

Prin logică, putem *formaliza* cerințele programelor

În limbaje *declarative* (de nivel înalt), putem specifica *ce* să facă programul, iar interpretorul/compilatorul efectuează calculul.

În încheiere: Calculabilitate

Ce se poate *calcula*, și cum putem defini această noțiune ?

Teza Church-Turing

o afirmație despre noțiunea de *calculabilitate*:
următoarele modele de calcul sunt echivalente:

- lambda-calculul
- mașina Turing
- funcțiile recursive

Lambda-calcul

Definit de Alonzo Church (1932); poate fi privit ca fiind cel mai simplu limbaj de programare

O expresie în lambda-calcul e fie:

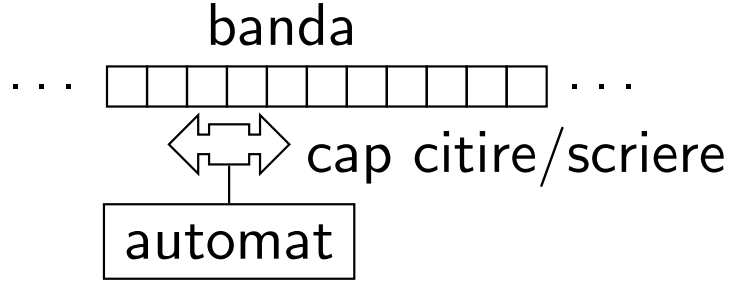
- o *variabilă* — x
- o *funcție* — $\lambda x\,.\,e$ (funcție de variabilă x)
  în ML: `fun x -> e`
- o *evaluare* de functie — $e_1\ e_2$ (funcția e_1 aplicată argumentului e_2)
  la fel în ML: `f 3` fără paranteze
  asociativă la stânga: `f x y` = `(f x) y`

Toate noțiunile fundamentale (numere naturale, booleni, perechi, etc.) pot fi exprimate în lambda-calcul.

Mașina Turing

Mașina Turing e compusă din:

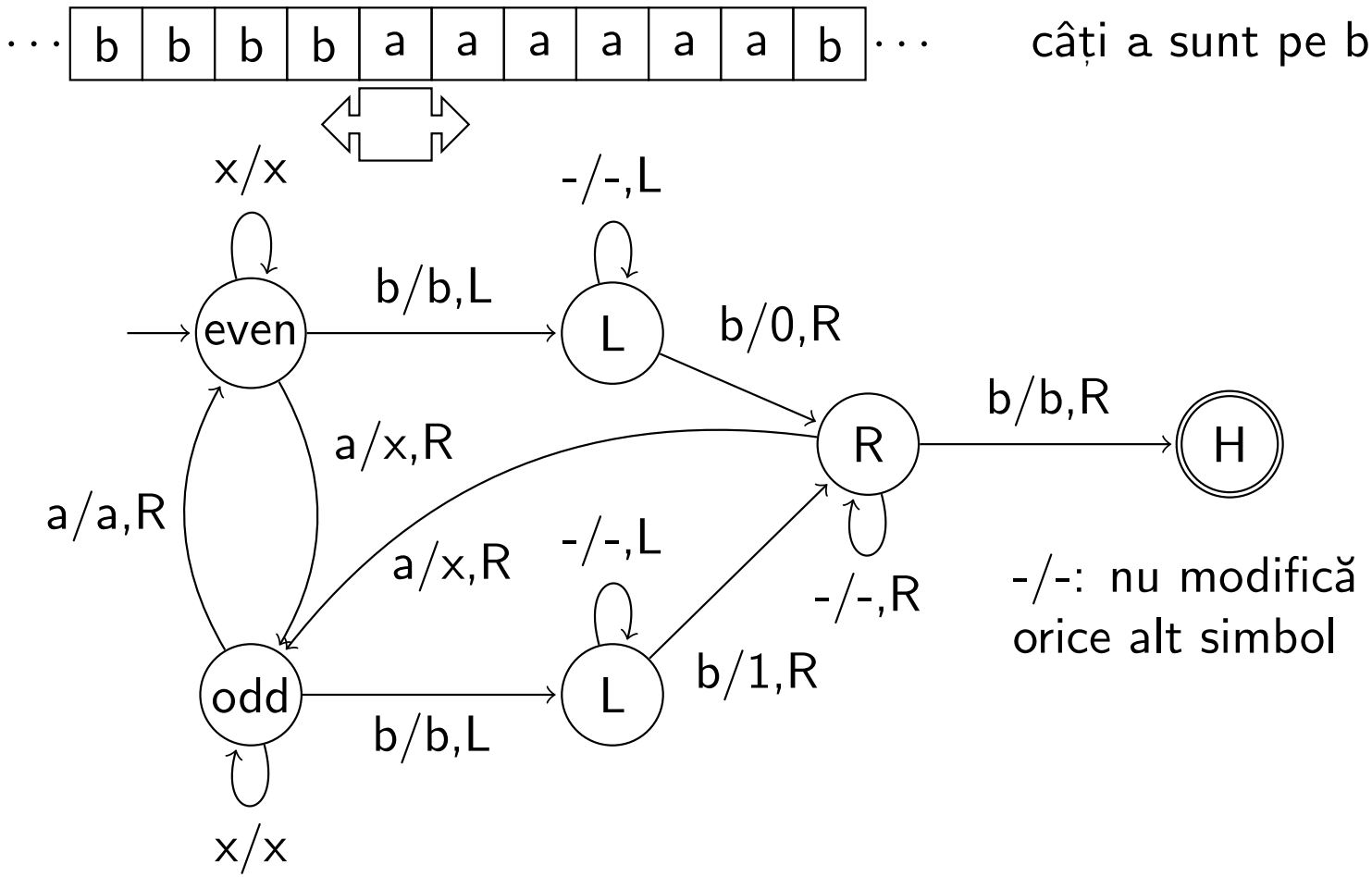
- o *bandă* cu un număr infinit de *celule*; fiecare conține un *simbol*
  (banda poate fi infinită la unul/ambele capete, e echivalent)
- un *cap* de citire/scriere, controlat de un *automat cu stări finite*



Automatul și conținutul benzii determină comportarea.
După 1) starea curentă și 2) simbolul aflat sub cap, mașina:
1) trece în starea următoare, 2) scrie un (alt) simbol sub cap
3) mută capul la stânga sau la dreapta

Inițial, banda are un șir finit de simboluri, capul e pe cel din stânga;
restul celulelor conțin un simbol special (numit vid sau blanc).

Exemplu: numără simboluri și scrie numărul în binar



obține fiecare bit din numărul de a
→: schimbă a cu x din doi în doi
←: scrie 0 sau 1 după paritate
repetă până nu mai sunt a: Halt

```
   bbbbaaaaaab → bbbbxaxaxab
← bbb0xaxaxab → bbb0xxxaxxb
← bb10xxxaxxb → bb10xxxxxxb
← b110xxxxxxb → b110xxxxxxb
Halt
```

Mașina Turing – descriere formală

Formal, mașina Turing se descrie printr-un tuplu cu 7 elemente:

Q: mulțimea stărilor automatului finit (de control)
Σ: mulțimea finită a *simbolurilor de intrare* (din șirul inițial)
Γ: mulțimea simbolurilor de pe bandă; $\Sigma \subset \Gamma$
$\delta : Q \times \Gamma \to Q \times \Gamma \times \{l, r\}$: funcția de tranziție:
dă starea următoare, simbolul cu care e înlocuit cel curent, și mutarea la stânga sau dreapta
(în unele versiuni, echivalente, capul poate și rămâne pe loc)
$q_0 \in Q$: starea inițială a automatului de control
$b \in \Gamma \setminus \Sigma$: simbolul vid (blanc): toate celulele cu excepția unui număr finit sunt inițial vide
$F \subseteq Q$: mulțimea stărilor finale, automatul se oprește (halt)

Poate descrie *orice calcul* (implementabil prin program)
Nu există algoritm care să decidă pentru orice automat și intrare dacă se oprește (*halting problem*) – la fel pentru programe

Calculabilitate și problema terminării (halting problem)

În formularea pentru programe:

Nu există algoritm (program) care ia un program arbitrar P și un set de date D și determină dacă $P(D)$ (rularea lui P cu datele D) s-ar termina (opri) sau ar rula la infinit.

Presupunem că ar exista un astfel de program *CheckHalt(P, D)*.
Deci, *CheckHalt*(X, X) spune ce face prog. X cu textul său ca date

Construim un "program imposibil" care face opusul a ceea ce face!

Întâi, definim programul *Test(X)* având ca intrare un program X:
dacă *CheckHalt*(X, X) decide `"halt"`, atunci **ciclează la infinit**
dacă *CheckHalt*(X, X) decide `"ciclează"`, atunci **stop**
Deci *CheckHalt*(X, X) spune ce face X(X) iar *Test*(X) face opusul

Se oprește *Test*(Test)? Răspunsul e dat de *CheckHalt*(Test,Test).
dar *Test*(Test) (cu X=*Test*) face *opusul* lui *CheckHalt*(Test,Test)
$\Rightarrow$ *contradicție*, deci nu poate exista *CheckHalt*!