Logică și structuri discrete

Complexitate și calculabilitate. Recapitulare

Marius Minea
marius@cs.upt.ro

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

8 ianuarie 2018

Revenim la traversarea grafurilor

Traversarea *prin cuprindere* pornind din nodul v_0
vizitează nodurile după distanță crescătoare de la v_0

Fie $N_k =$ mulțimea nodurilor cu drum de lungime $\leq k$ de la v_0

$N_0 = \{v_0\}$	doar nodul inițial
$N_1 = N_0 \cup vecini(N_0)$	nodul inițial și vecinii lui
$N_2 = N_1 \cup vecini(N_1)$	nodurile cu drum ≤ 1 și vecinii lor

...

Fie $f(S) = S \cup vecini(S)$. Atunci $N_{k+1} = f(N_k)$

Punctul fix

x e punct fix pentru o funcție f dacă $f(x) = x$

Dacă aplicăm funcția repetat:
pentru un punct fix, nu se mai produce o transformare

În cazul nostru, $f(S) = S \cup \mathit{vecini}(S)$, avem $S \subseteq f(S)$
repetând: $S \subseteq f(S) \subseteq f(f(S)) \subseteq \dots$

Mulțimea nodurilor e *finită*, deci la un moment dat, șirul nu mai poate crește: $\exists n \,.\, f^n(N_0) = f^{n+1}(N_0)$
am vizitat toate nodurile accesibile din v_0

Putem aplica parcurgerea până la punct fix și fără să construim explicit un graf, pentru stările / configurațiile oricărui sistem (mutări în jocuri, valori calculate în programe, etc.)

Complexitate și calculabilitate

Scriind programe, ne punem întrebarea fundamentală:

Dată fiind o problemă, se poate scrie un program care o rezolvă?

Se poate scrie un antivirus perfect ?
(detectează toți virușii, fără alerte false)

Se poate scrie compilatorul care generează codul optim?

Dacă problema e rezolvabilă, ne întrebăm și:

Cât de complexă e soluția?

Complexitatea algoritmică

Efortul de rezolvare depinde (de regulă) de dimensiunea intrării:

Căutare într-o listă neordonată (de lungime n)
proporțională cu n (lungimea listei)

Căutare într-un arbore binar de căutare (cu n noduri)
proporțională cu înălțimea arborelui
$\log_2 n$ dacă arborele e echilibrat

Sortarea unui vector de n elemente
proporțională cu $n \log_2 n$ pentru mergesort (sau quicksort, cu selecția corespunzătoare a medianei)

O problemă are *complexitatea* $O(f(n))$ (litera O mare, "big-O")
în exemplele date: $O(n)$, $O(\log n)$, $O(n \log n)$
dacă există $c > 0$ astfel încât soluția ia $\leq c \cdot f(n)$ pași pentru n suficient de mare ($n \geq n_0$ dependent de problemă)

Clasele de complexitate P și NP

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

Exemplu: realizabilitatea formulelor boolene
O formulă cu n propoziții are 2^n atribuiri
$\Rightarrow$ timp *exponențial* încercând toate

O atribuire dată se verifică în timp *liniar* (în dimensiunea formulei)
parcurgem formula o dată și obținem valoarea
$\Rightarrow$ realizabilitatea e în NP

dar nu se cunoaște un algoritm polinomial (din câte știm nu e în P)

În general, a *verifica* o soluție e (mult) mai simplu decât a o *găsi*.

Probleme NP-complete

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 ⇒ ar fi P = NP (se crede P ≠ NP)

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

problema colorării grafurilor
(câte culori astfel ca noduri adiacente să fie de culori diferite?)

problema rucsacului
(selecție de obiecte de valoare maximă, cu greutate totală limitată)

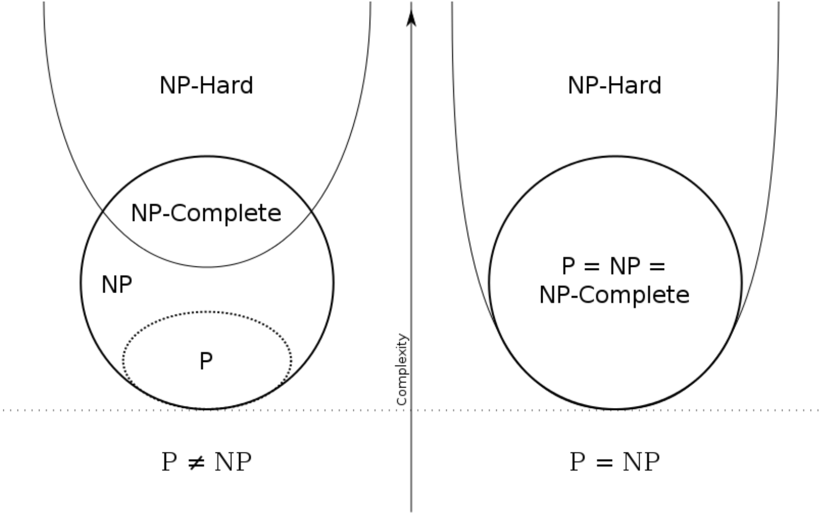
"sumset-sum":
într-o mulțime de întregi există o submulțime de sumă dată?

Cum demonstrăm că o problemă e NP-completă (grea) ?

reducem o problemă cunoscută din NP la problema studiată
⇒ 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`

Ce se poate și nu se poate calcula?

Revenim la întrebarea

Dată fiind o problemă, se poate scrie un program care o rezolvă?

Teorema lui Cantor: *Nu există bijecție* de la X la $\mathcal{P}(X)$

$$|X| < |\mathcal{P}(X)|$$

Care e legătura?

Programe și probleme

Un *program* (în cod sursă) e un șir de caractere.
deși nu orice șir de caractere e un program valid

Notând cu *Progs* mulțimea programelor și Σ alfabetul caracterelor

$$Progs \subseteq \Sigma^*$$

O clasă de *probleme* (sunt multe altele):
Fiind dată o mulțime (finită sau nu) de șiruri $S = \{s_1, s_2, ..., s_n, ...\}$, și un șir w, aparține el mulțimii date, $w \in S$?

Mulțimea S ar putea fi dată: explicit, printr-o expresie regulată, un automat, o gramatică, ...
Orice mulțime de șiruri definește (cel puțin) o problemă

$$\mathcal{P}(\Sigma^*) \subseteq Probs$$

Teorema lui Cantor ne spune atunci:

$$|Progs| \leq |\Sigma^*| < |\mathcal{P}(\Sigma^*)| \leq |Probs|$$

Nu putem asocia deci fiecărei probleme un program!

Limitările logicii

Am văzut că nu se poate calcula orice.

Însă, teoretic, se poate demonstra (sau infirma) orice afimație matematică?

Demonstrație (sintactică) și consecință logică (semantică)

În logica predicatelor, numărul interpretărilor e *infinit*
$\Rightarrow$ nu putem construi un tabel de adevăr pentru o formulă

E *esențial* deci să putem *demonstra* (deduce) o formulă.

Deducția $H \vdash \varphi$ a formulei φ din ipotezele H e pur sintactică:
un șir de formule, fiecare o *axiomă* sau o *ipoteză* sau rezultând printr-o *regulă de inferență* (modus ponens) din formule anterioare

Consecința semantică/implicația logică e o noțiune semantică, considerând *interpretări* și valori de adevăr:

Ipotezele 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 *consistent* care poate exprima aritmetica elementară e incomplet

i.e., se pot scrie afirmații care nu pot fi *nici demonstrate nici infirmate* în acel sistem

în particular, 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)

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

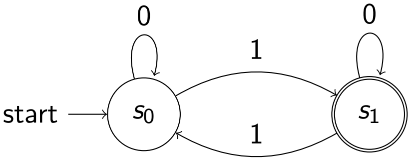
Definit de Alonzo Church (1932); poate fi privit ca fiind cel mai simplu limbaj de programare

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 x`	fără paranteze

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

Un alt model de calcul: automatul



Automatul de mai sus *calculează* paritatea unui șir de 0 și 1
(care la rândul său, poate reprezenta un număr în binar)
are un număr par sau impar de 1 ?

Comportamentul e determinat complet de *stare* și *intrare*

Automatul "știe" doar starea în care se află:
are *memorie finită*

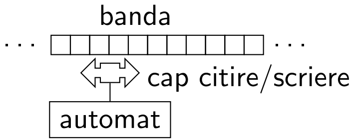
Dacă are n stări, am putea reprezenta starea ca o valoare pe $\lceil \log_2 n \rceil$ biți

Cum reprezentăm însă un calculator, care (conceptual) nu are limită de memorie?

Mașina Turing

Mașina Turing e compusă din:

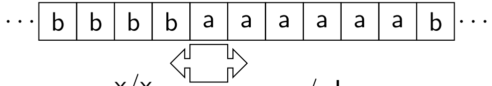
- o *bandă* cu un număr infinit de *celule*; fiecare conține un *simbol*
  (banda poate fi infinită la unul/ambele capete, e echivalent)
- un *cap* de citire/scriere, controlat de un *automat cu stări finite*



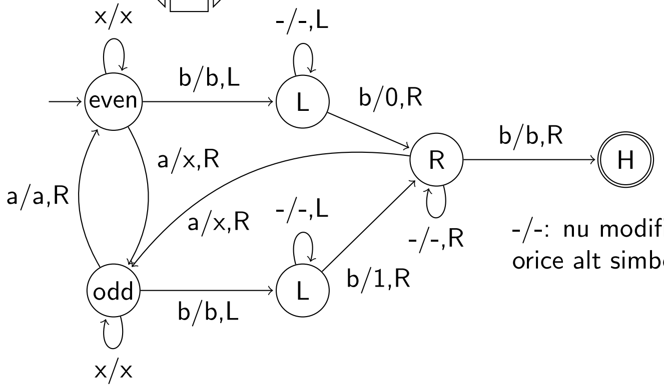
Automatul și conținutul benzii determină comportarea.
După 1) starea curentă și 2) simbolul aflat sub cap, mașina:
1) trece în starea următoare, 2) scrie un (alt) simbol sub cap
3) mută capul la stânga sau la dreapta

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



câți a sunt pe bandă?



-/-: nu modifică
orice alt simbol

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)
⇒ *contradicție*, deci nu poate exista *CheckHalt*!

Recapitulare: noțiuni de programare

Funcțiile sunt valori ("first-class values") care pot fi manipulate la fel ca orice alte valori:
pot fi transmise ca argumente, returnate ca rezultate)

În C: putem transmite / returna / atribui *pointeri* la funcții
ex. funcția de comparare la `qsort`

Tipuri

ML *verifică static* (la compilare) corectitudinea tipurilor
elimină multe erori încă de la compilare

Inferența de tipuri: tipurile nu trebuie precizate explicit, ele sunt deduse de compilator (din operațiile folosite)

Polimorfism: funcții care pot opera pe familii de mai multe tipuri
(liste, arbori, etc. de tipuri arbitrare)

Definirea de tipuri

Tipuri compuse:

produs cartezian: tuple (perechi, triplete)
similar structurilor în C (și în ML, câmpurile pot avea nume)

reuniune disjunctă: tipurile cu variante

```
type 'a tree = L of 'a | T of 'a tree * 'a * 'a tree
```

Potrivirea de tipare
mecanism puternic pentru lucrul cu tipuri compuse
compilatorul verifică tratarea tuturor cazurilor

Gestiunea memoriei

Funcțiile pot *returna orice valori* compuse

Nu e necesară alocarea dinamică explicită
și nici eliberarea memoriei
gestionată automat la rulare ("garbage collection")

C distinge între valori "obișnuite" care pot fi returnate
(întregi, reali, structuri)
și valori reprezentate prin adresa lor de memorie
(tablouri, șiruri, funcții, ...)

Stilul funcțional vs. imperativ

Funcțiile calculează valori.

Întreg programul e o *expresie*

Secvențierea e necesară/utilă doar pentru scriere la ieșire

ATENȚIE!
`expr1; expr2; expr3`
ignoră rezultatele primelor două expresii
⇒ are sens doar dacă acestea tipăresc valori
(și la citire, valoarea trebuie transmisă funcției de prelucrare)

Abstracția

ML are *module* care separă *interfața* de *implementare*

Avem funcții pentru mulțimi, asocieri, etc.
fără a fi expuse detaliile de reprezentare

Importantă în toate paradigmele de programare

Exemplu: în matematică, o mulțime poate fi dată
   prin enumerarea elementelor
   printr-o proprietate:
      : interval de valori: [a, b]
      : constrângeri: $x \leq 5 \wedge y - x \leq 3$
   : funcția caracteristică / de membru în mulțime

Algoritmii în probleme care lucrează cu mulțimi pot fi scriși
independent de reprezentare!