Complexitatea algoritmilor

Ștefan Ciobâcă, Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
stefan.ciobaca@info.uaic.ro, dlucanu@info.uaic.ro

PA 2015/2016

Plan

Problemă computațională

O problemă propusă pentru a fi rezolvată de un algoritm poate fi reprezentată prin:

- domeniul problemei
- o pereche $(input, output)$

Notație:

$p \in P \equiv p$ *instanță* (componenta *input*) a lui P

$P(p) \equiv$ *rezultatul* (componenta *ouput* a) lui P pentru p

Exemplu: problema Platou 1/2

Domeniul problemei:

- considerăm secvențe $a = (a_0, \ldots, a_{n-1})$ de numere întregi
- segment $a[i..j]$: $(a_i, \ldots, a_j)$, unde $i \leq j$
- dacă $i > j$, $a[i..j]$ este secvența vidă
- lungimea unui segment $a[i..j]$ este $j + 1 - i$
- platou este un segment cu toate elementele egale

Input: O secvență $a = (a_0, \ldots, a_{n-1})$ de numere întregi de lungime n ordonată crescător.

Output: Lungimea celui mai lung platou.

Exemplu: problema Platou 2/2

Perechea $(input, output)$ reprezentată cu ajutorul predicatelor:

$platou(a, i, j)$: $(\forall k) i \leq k \leq j \implies a_i = a_k$
$ordonatCrescator(a)$: $a_0 \leq \ldots \leq a_{n-1}$
$ordonatCrescator(a) \implies (platou(a, i, j) \iff a_i == a_j)$

$Input$: $a == (a_0, \ldots, a_{n-1}) \wedge ordonatCrescator(a)$.

$Output$: $q \in \mathbb{Z} \wedge$
$(\exists 0 \leq i \leq j < n) platou(a, i, j) \wedge q = j + 1 - i \wedge$
$(\forall 0 \leq k \leq \ell < n) platou(a, k, \ell) \implies q \geq (\ell + 1 - k)$.

$input \equiv$ precondiție
$output \equiv$ postcondiție
$(precondiție, postcondiție) \equiv$ specificație

Problemă rezolvată de un algoritm

A rezolvă o problemă *P* dacă:

- $(\forall p \in P)\ (\exists \langle A, \sigma_p \rangle)$ a.î. σ_p include structuri date ce descrie p;
- $\langle A, \sigma \rangle \Rightarrow^* \langle ., \sigma' \rangle$; și
- σ' include structuri de date ce descriu $P(p)$.

Problemă rezolvată de un algoritm, mai formal

aserțiunie ϕ: formulă cu predicate descrisă cu variabilele care apar în algoritm

$\sigma \models \phi$: valorile variabilelor în σ satisfac ϕ
Exemplu: dacă $\sigma = \mathtt{x} \mapsto 3\ \mathtt{y} \mapsto 5$, atunci $\sigma \models 2 * \mathtt{x} > \mathtt{y}$ și $\sigma \not\models \mathtt{x} + \mathtt{y} < 0$.

P este specificată de $(pre, post)$ (i.e., $(precondiție, postcondiție)$)

A rezolvă $P \equiv (\forall \sigma)$ cu $\sigma \models pre$ $(\exists \sigma')$ a.î. $\langle A, \sigma \rangle \Rightarrow^* \langle ., \sigma' \rangle$ și $\sigma' \models post$

Rezolvare versus Corectitudine

A este corect $\equiv$ A rezolvă o problemă specificată prin (*precondiție*,*postcondiție*)

Cele două noțiuni nu sunt chiar echivalente.

Există două tipuri de corectitudine:

corectitudine totală: $(\forall\sigma)$, dacă $\sigma \models pre$ atunci $(\exists\sigma')$ a.î. $\langle A, \sigma\rangle \Rightarrow^* \langle ., \sigma'\rangle$ și $\sigma' \models post$

corectitudine parțială: $(\forall\sigma)$, dacă $\sigma \models pre$ și dacă $(\exists\sigma')$ a.î. $\langle A, \sigma\rangle \Rightarrow^* \langle ., \sigma'\rangle$, atunci $\sigma' \models post$

Rezolvarea este echivalentă cu corectitudinea totală.

Algoritmul PlatouAlg

Pp. că secvența a este reprezentată de tabloul $\mathtt{a} \mapsto [a_0 \ldots a_{n-1}]$

Un algoritm care rezolvă problema Platou:

```
lg = 1;
i = 1;
while (i < n) {
  if (a[i] == a[i - lg]) lg = lg+1;
  i = i + 1;
}
```

Relația dintre PlatouAlg și specificarea problemei Platou

orice execuție care pleacă din configurația inițială:

$\langle PlatouAlg,\ \mathtt{n} \mapsto n\ \mathtt{a} \mapsto [a_0, \ldots, a_{n-1}] \rangle$

cu $a_0 \leq \cdots \leq a_{n-1} \wedge n \geq 1$ (i.e., satisface precondiția)

se oprește în configurația finală:

$\langle \cdot,\ \mathtt{n} \mapsto n\ \mathtt{a} \mapsto \{0 \mapsto a_0 \ldots n-1 \mapsto a_{n-1}\}\ \mathtt{i} \mapsto n\ \mathtt{lg} \mapsto q \rangle$

și q reprezintă lungimea celui mai lung platou din a:

$$(\exists 0 \leq i \leq j < n) platou(a, i, j) \wedge q = j + 1 - i \wedge$$
$$(\forall 0 \leq k \leq \ell < n) platou(a, k, \ell) \implies q \geq (\ell + 1 - k)$$

(i.e., satisface postcondiția)

Cum demonstrăm corectitudinea?

Cum dovedim că algoritmul PlatouAlg rezolvă într-adevăr problema Platou?

O posibilă soluție:

- la începutul și la sfârșitul buclei while:

lg reprezintă lungimea celui mai lung platou din segmentul $a[0..i-1]$, i.e.

$$(\exists 0 \leq i_0 \leq j_0 < i) platou(a, i_0, j_0) \wedge \texttt{lg} = j_0 + 1 - i_0 \wedge$$
$$(\forall 0 \leq k \leq \ell < i) platou(a, k, \ell) \implies \texttt{lg} \geq (\ell + 1 - k)$$

- această prorietate se numește invariant de buclă

- la sfârșitul buclei: invariantul și negația condiției $(i \geq n)$

invariantul și $i = n \implies$ postcondiția

deci trebuie arătat că și $i \leq n$ este invariant

Cum demonstrăm invariantul?

Distingem două cazuri:

1. $j_0 = i - 1$ (cel mai lung platou din $a[0..i-1]$ se termină în $i - 1$).
Distingem două subcazuri:

1.1 are loc $platou(a, i_0, i)$ (i.e. `a[i] == a[i - lg]`)

1.2 nu are loc $platou(a, i_0, i)$

2. cel mai lung platou din $a[0..i-1]$ NU se termină în $i - 1$

Problemă rezolvabilă (calculabilă)

O problemă P este rezolvabilă (calculabilă) dacă există un algoritm A care rezolvă P.

O problemă P este nerezolvabilă (necalculabilă) dacă NU există un algoritm A care rezolvă P.

Problemă de decizie

Problemă de decizie: răspunsul (outputul) este de forma "DA" sau "NU" (echivalent, "true" sau "false")

Reprezentarea unei probleme de decizie:

O problemă decidabilă este o problemă de decizie rezolvabilă.

O problemă nedecidabilă este o problemă de decizie nerezolvabilă.

Sunt toate problemele computaționale rezolvabile (decidabile)?

La începutul secolului 20, matematicienii credeau că da.

În 1931, Kurt Gödel a șocat dovedind că aceasta este imposibil.

> *Oricât de puternic ar fi un sistem de raționament matematic, vor exista afirmații care nu pot fi demonstrate.*

(Celebra teorema de incompletitudine alui Gödel).

Câțiva ani mai târziu, Alan Turing a demonstrat același lucru utilizând noțiunea de algoritm (mașină Turing).

Program universal

program (algoritm) universal:
– intrare: un program (algoritm) A și o intrare x (echivalent, o configurație $\langle A, \sigma_x \rangle$)
– comportare: simulează activitatea lui A pentru intrarea x

Programele (algoritmii) pot fi intrări pentru alți algoritmi!

Exemplu de problemă nerezolvabilă/nedecidabilă

Problema opririi:
Instance: O configurație $\langle A, \sigma_0 \rangle$, unde A este un algoritm și σ_0 codificarea unei intrări pentru A.
Question: Execuția care pleacă din configurația inițială $\langle A, \sigma_0 \rangle$ este finită?

Teoremă

Nu există un algoritm care să rezolve Problema opririi.

Ideea de demonstrare 1/2

Prin reducere la absurd.

Presupunem că există un algoritm H care rezolvă problema opririi.

Construim un alt aloritm, care apelează H:

```
NewH(A) {
  if H(A, A) return true;
  else return false;
}
```

și un alt program care apelează NewH:

```
HaltsOnSelf (A) {
  if NewH(A) while (true) {}
  else return false;
}
```

Ideea de demonstrare 2/2

Ce se întâmplă când se apelează `HaltsOnSelf(HaltsOnSelf)`?

Execuția lui `HaltsOnSelf(HaltsOnSelf)` nu se termină; rezultă că `NewH(HaltsOnSelf)` întoarce `true`, care implică `H(HaltsOnSelf,HaltsOnSelf)` întoarce `true`. Contradicție.

Execuția lui `HaltsOnSelf(HaltsOnSelf)` se termină și întoarce `true`; rezultă că `NewH(HaltsOnSelf)` întoarce `false`, care implică `H(HaltsOnSelf,HaltsOnSelf)` întoarce `false`. Contradicție din nou.

Rezolvarea teoremei de mai sus este strâns legată de următorul paradox logic. "Există un oraș cu un bărbier care bărbierește pe oricine ce nu se bărbierește singur. Cine bărbierește pe bărbier?"

Alte exemple de probleme nedecidabile

Problema echivalenței programelor.

Totality: dacă un program dat se oprește pentru toate intrările.

Problema corectitudinii totale.

Problema a 10-a a lui Hilbert

...

Parțial rezolvabil (calculabil, decidabil)

O problemă de decizie este parțial calculabilă (semidecidabilă) dacă există un algoritm care se oprește cu răspunsul "DA" pentru toate intrările pentru care răspunsul correct este "DA".

Este Problema opririi semidecidabilă?

Plan

Timpul unei execuții

Fie $E = \langle A_0, \sigma_0 \rangle \Rightarrow \cdots \Rightarrow \langle A_n, \sigma_n \rangle$ o execuție. Timpul consumat de această execuție este suma timpilor pașilor de execuție:

$$time_d(E) = \sum_{i=0}^{n-1} time_d(\langle A_i, \sigma_i \rangle \Rightarrow \langle A_{i+1}, \sigma_{i+1} \rangle)$$

unde $d \in \{log, unif\}$

Demo cu versiunea de Alk care calculează și timpii uniform și logaritmic.

Timpul necesar execuției unei instanțe

Algoritm determinist: $\forall \langle A_i, \sigma_i \rangle$ accesibilă din configurația inițială $\langle A_0, \sigma_0 \rangle$, $\exists$ cel mult o $\langle A', \sigma' \rangle$ cu $\langle A_i, \sigma_i \rangle \Rightarrow \langle A', \sigma' \rangle$.

Fie P o problemă rezolvată de A.

$\forall p \in P$ există un calcul unic $E_p = \langle A, \sigma_p \rangle \Rightarrow \cdots$.

Timpul necesar algoritmului A pentru a rezolva instanța p este

$$time_d(A, p) = time_d(E_p)$$

unde $d \in \{log, unif\}$

Plan

Dimensiunea unei instanțe

Dimensiunea unei stări σ:

$$size_d(\sigma) = \sum_{\mathtt{x} \mapsto v \in \sigma} size_d(v)$$

Dimensiunea unei configurații:

$$size_d(\langle A, \sigma \rangle) = size_d(\sigma)$$

Fie P o problemă rezolvată de A.

Dimensiumea lui $p \in P$:

$$size_d(p) = size_d(\langle A, \sigma_p \rangle) \quad (= size(\sigma_p))$$

unde $d \in \{log, unif\}$.

Complexitatea timp în cazul cel mai nefavorabil

Fie P o problemă și A un algoritm determinist care rezolvă P și $d \in \{log, unif\}$.

Grupăm instanțele p ale problemei P în clase de echivalență: p și p' sunt în aceeași clasă dacă $size_d(p) = size_d(p')$.

Un număr întreg pozitiv n poate fi privit ca fiind clasa de ecivalență a instanțelor de mărime n.

Complexitatea timp în cazul cel mai nefavorabil:

$$T_{A,d}(n) = \max\{time_d(A, p) \mid p \in P, size_d(p) = n\}$$

Complexitatea spațiu

Fie $E = \langle A_0, \sigma_0 \rangle \Rightarrow \cdots \Rightarrow \langle A_n, \sigma_n \rangle$ o execuție și $d \in \{log, unif\}$.

Spațiul consumat de această execuție este dat de maximul dintre dimensiunile configurațiilor din E:

$$space_d(E) = \max_{i=0}^{n} size_d(\langle A_i, \sigma_i \rangle)\}$$

Spațiul necesar algoritmului A pentru a rezolva instanța p este

$$space_d(A, p) = space_d(E_p)$$

Complexitatea spațiu în cazul cel mai nefavorabil este calculată într-un mod similar complexității timp pentru cazul cel mai nefavorabil:

$$S_{A,d}(n) = \max\{space_d(A, p) \mid size_d(p) = n\}$$

Calcul complexității în cazul cel nefavorabil 1/3

- A este o expresie E care nu include apeluri de funcții (algoritmi):
  $T_{A,d}(n) = time_d([\![E]\!](\sigma_p))$ pentru $p \in P$ cu $size_d(p) = n$
- A este o atribuire $X = E$;
  $T_{A,d}(n) = T_{E,d}(n)$
- A este `if (`E`)` S_1 `else` S_2:
  $T_{A,d}(n) = \max\{T_{S_1,d}(n), T_{S_2,d}(n)\} + T_{E,d}(n)$
- A o compunere secvențială S_1 S_2:
  $T_{A,d}(n) = T_{S_1,d}(n) + T_{S_2,d}(n)$

Calcul complexității în cazul cel nefavorabil 2/3

- A este o instrucțiune iterativă (e.g.,`while, for`): de multe ori se poate calcula doar o aproximare
  - soluția 1 (o aproximare mai fidelă):
    - se calculează numărul maxim de iterații $nMax$
    - se calculează complexitatea în cazul cel nefavorabil pentru fiecare iterație, fie acestea $T_1, \ldots, T_{nMax}$
    - se ia $T_{A,d}(n) = T_1 + \ldots + T_{nMax}$
  - soluția 2 (approximare mai grosieră):
    - se calculează numărul maxim de iterații $nMax$
    - se calculează complexitatea în cazul cel nefavorabil pentru iterația cu timpul (în cazul cel nefavorabil) cel mai mare, fie acesta T_{itMax}
    - se ia $T_{A,d}(n) = nMax \times T_{itMax}$

Calcul complexității în cazul cel nefavorabil 3/3

- Atenție la liste, mulțimi, . . . :

```
s = 0;
for(i = 0; i < l.size(); ++i)  // l is a linear list
  s = s + l.at(i);
```

```
s = emptySet;
forall x in a   // a is a set
  if (x % 2 == 0) s = s U singletonSet(x);
```

- Apel de funcții (algoritmi):
  - se estimează dimensiunea argumentelor în funcție de dimensiunea instanței n
  - se utilizează complexitatea în cazul cel nefavorabil a algoritmului apelat, calculată cu dimensiunea argumentelor estimată

Calculul complexității în cazul cel nefavorabil în practică

- deobicei numai costul uniform este calculat
- trebuie precizată dimensiunea unei instanțe
- numai o parte din operații sunt considerate (e.g., comparări, atribuiri)
- cel mai important (și uneori) dificil este identificarea cazului cel mai nefavorabil
- se calculează aproximații ale lui $T_{A,d}(n)$ utilizând notațiile $O(f(n))$, $\Omega(f(n))$,$\Theta(f(n))$

Reamintim:

$$O(f(n)) = \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \leq c \cdot |f(n)|\}$$
$$\Omega(f(n)) = \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \geq c \cdot |f(n)|\}$$
$$\Theta(f(n)) = \{g(n) \mid (\exists c_1, c_2 > 0, n_0 \geq 0)(\forall n \geq n_0)c_1 \cdot |f(n)| \leq |g(n)| \leq c_2 \cdot |f(n)|\}$$

Exemplul 1

input: $n, (a_0, \ldots, a_{n-1}), z$ numere întregi.

output: $poz = \begin{cases} \min\{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$

```
i = 0;
while (a[i] != z) and (i < n-1)
  i = i+1;
if (a[i] == z) poz = i;
else poz = -1;
```

Discuția pe tablă.

Exemplul 2

input: $n, (a_0, \ldots, a_{n-1})$ numere întregi.
output: $max = \max\{a_i \mid 0 \leq i \leq n-1\}$.

```
max = a[0];
for (i = 1; i < n; i++)
  if (a[i] > max)
    max = a[i];
```

Discuția pe tablă.

Exemplul 3

input: $n, (a_0, \ldots, a_{n-1})$ numere întregi.
output: $(a_{i_0}, \ldots, a_{i_{n-1}})$ unde $(i_0, \ldots, i_{n-1})$ este o permutare a șirului $(0, \ldots, n-1)$ și $a_{i_j} \leq a_{i_{j+1}}, \forall j \in \{0, \ldots, n-2\}$.

```
for (k = 1; k < n;  k++) {
  temp = a[k];
  i = k - 1;
  while (i >= 0 and a[i] > temp) {
    a[i+1] = a[i];
    i = i-1;
  }
  a[i+1] = temp;
}
```

Discuția pe tablă.

Exemplul 4

input: $n, (a_0, \ldots, a_{n-1}), z$ numere întregi;
secvența $(a_0, \ldots, a_{n-1})$ este sortată crescător,

output: $poz = \begin{cases} k \in \{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$

```
  istg = 0;
  idr = n - 1;
  while (istg <= idr ) {
    imed = (istg + idr) / 2;
    if (a[imed] == z)
      return imed
    else if (a[imed] > z)
      idr = imed-1;
    else
      istg = imed + 1;
  }
return -1
```

Discuția pe tablă.