

Outline

Cuprins

1 Complexitatea problemelor

De ce definim complexitatea unei probleme

Până acum am clasificat problemele în rezolvabile și nerezolvabile.

Pentru o problemă rezolvabilă pot exista mai mulți algoritmi care să o rezolve.

De fapt dacă există unul, atunci există o infinitate. (De ce?)

Am văzut cum se măsoară eficiența unui algoritm.

Ce putem spune despre eficiența rezolvării unei probleme?

Definițiile de la eficiența algoritmilor pot fi ușor transferate la probleme. De exemplu, complexitatea timp a unei probleme se referă la complexitatea timp a algoritmilor care rezolvă problema. Pentru fiecare versiune a definiției pentru algoritmi (în cazul cel mai nefavorabil, medie, cost uniform, cost logaritm, ...), vom avea una corespunzătoare pentru probleme.

Definiția complexității $O(f(n))$ a unei probleme

Oferă o margine superioară pentru efortul computațional necesar rezolvării unei probleme.

Definition

Problema P are complexitatea timp în cazul cel mai nefavorabil $O(f(n))$ dacă există un algoritm A care rezolvă P și $T_A(n) = O(f(n))$.

Pentru a arăta că o problemă P are complexitatea timp în cazul cel mai nefavorabil $O(f(n))$, este suficient de găsit un algoritm A care rezolvă P și să arătăm că A are complexitatea timp în cazul cel mai nefavorabil $O(f(n))$. Valoarea lui $f(n)$ ne dă o margine superioară pentru timpul necesar rezolvării unei instanțe de dimensiune n .

Definiția complexității $\Omega(f(n))$ a unei probleme

Oferă o margine inferioară pentru efortul computațional necesar rezolvării unei probleme.

Definition

P are complexitatea timp în cazul cel mai nefavorabil $\Omega(f(n))$ dacă orice algoritm A care rezolvă P are $T_A(n) = \Omega(f(n))$.

Acest tip de informație este mult mai dificil de obținut deoarece trebuie arătat că nu există algoritmi care să rezolve orice instanță de dimensiune n într-un timp mai mic decât $f(n)$ multiplicat cu o constantă. Vom studia doar două probleme pentru care există dovedită această margine inferioară: sortarea și căutarea.

Algoritm optim pentru o problemă

Definition

A este algoritm optim (din punct de vedere al complexității timp pentru cazul cel mai nefavorabil) pentru problema P dacă

- A rezolvă P și
- P are complexitatea timp în cazul cel mai nefavorabil $\Omega(T_A(n))$.

Se poate dovedi că un algoritm este optim din punct de vedere al timpului de execuție numai dacă se cunoaște limita inferioară pentru problemă; din acest motiv se cunosc puțini algoritmi optimi.

2 Complexitatea sortării

Problema sortării

Considerăm cazul particular al sortării tablourilor:

SORT

Input n și tabloul $a = [v_0, \dots, v_{n-1}]$.

Output tabloul $a' = [w_0, \dots, w_{n-1}]$ cu proprietățile: $w_0 \leq \dots \leq w_{n-1}$ și $w = (w_0, \dots, w_{n-1})$ este o permutare a secvenței $v = (v_0, \dots, v_{n-1})$.

Notatii:

$SORTED(a)$: tabloul a este sortat, i.e. $a[0] \leq \dots \leq a[n-1]$

$Perm(v, w)$: w este o permutare a lui v

Sortare prin interschimbare (BubbleSort) 1/2

Metoda bubble sort se bazează pe următoarea definiție a predicatului $SORTED(a)$:

$$SORTED(a) \iff (\forall i)(0 \leq i < n-1) \Rightarrow a[i] \leq a[i+1]$$

unde $n = a.size()$. (Aceasta e parte a **domeniului problemei**.)

Dacă $a[i] > a[i+1]$, spunem că perechea $(i, i+1)$ formează o **inversiune**.

În cazul unei inversiuni proprietatea $a[i] \leq a[i+1]$ poate fi stabilită printr-o interschimbare. De aici derivă foarte simplu un algoritm care restabilește relația de ordine corectă între elementele care formează inversiuni:

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

Sortare prin interschimbare (BubbleSort) 2/2

Procesul de restabilire de mai sus trebuie repetat până nu mai sunt inversiuni:

```
while (posibil să mai existe inversiuni) {
  for (i=0; i < n-1; ++i) {
    if (a[i] > a[i+1]) {
      swap (a, i, i+1);
    }
  }
}
```

(Acesta este pseudocod!)

Segmentul din tablou de la sfârșit care nu include versiuni la o ierație while, nu va include inversiuni nici la iterațiile următoare. De aici rezultă ca testul *nu mai există inversiuni* poate fi verificat ținând minte poziția ultimei inversiuni.

BubbleSort: algoritmul

```
bubbleSort(a, n) {
    ultim = n-1;
    while (ultim > 0) {
        n1 = ultim;
        ultim = 0;
        for (i=0; i < n1; ++i) {
            if (a[i] > a[i+1]) {
                swap(a, i, i+1);
                ultim = i;
            }
        }
    }
}
```

Evaluarea algoritmului BubbleSort 1/2

Corectitudine

Invariant bucla while: $a[ultim+1 .. n-1]$ include cele mai mari $n-1-ultim$ elemente din a inițial (v_0, \dots, v_{n-1}) ordonate crescător (i.e., avem $SORTED(a[ultim+1 .. n-1])$)

Invariant bucla for: $a[j] \leq a[i]$ pentru $j = 0, \dots, i$.

Singura instrucțiune care modifică tabloul a este **swap** și aceasta menține proprietatea $Perm(u, u')$, unde u este valoarea variabilei a înainte de **swap** și u' cea de după.

Evaluarea algoritmului BubbleSort 1/2

Timp de execuție

- dimensiune instanță: n ($= a.size()$)
- operații măsurate: comparațiile care implică elementele tabloului
- cazul cel mai nefavorabil: când secvența de intrare este ordonată descrescător
- numărul de comparații pentru acest caz este $(n-1) + (n-2) + \dots + 1 = \frac{(n-1)n}{2} = O(n^2)$

Sortare prin inserție directă (InsertSort) 1/2

Principiul de bază al algoritmului de sortare prin inserție este următorul: Se presupune că subsecvența $a[0 .. j-1]$ este sortată. Se caută în această subsecvență locul i al elementului $a[j]$ și se inserează $a[j]$ pe poziția i . Procesul de mai sus trebuie repetat pentru $j = 1, \dots, n-1$:

```
for (j=1; j < n; ++j )
    inserează a[j] în a[0..j-1] a.î. SORTED(a[0..j])
```

(Acesta este un pseudocod!)

Sortare prin inserție directă (InsertSort) 2/2

Analiza domeniului problemei

Poziția i pe care trebuie inserat $a[j]$ este determinată astfel:

- $i = j$ dacă $a[j] \geq a[j-1]$;
- $i = 0$ dacă $a[j] < a[0]$;
- $0 < i < j$ și satisface $a[i-1] \leq a[j] < a[i]$

Așadar elementele de pe pozițiile $i..j-1$ trebui deplasate la dreapta cu o poziție. Condiția pentru deplasarea la dreapta este $i \geq 0 \wedge a[i] > a[j]$ ($a[j]$ cel inițial):
Algoritm:

```
i = j - 1;
temp = a[j];
while ((i >= 0) && (a[i] > temp)) {
    a[i+1] = a[i];
    i = i - 1;
}
```

InsertSort: algoritmul

```
insertSort(a, n) {
    for (j = 1; j < n; ++j) {
        i = j - 1;
        temp = a[j];
        while ((i >= 0) && (temp < a[i])) {
            a[i+1] = a[i];
            i = i - 1;
        }
        if (i != j-1) a[i+1] = temp;
    }
}
```

Evaluarea algoritmului InsertSort 1/2

Corectitudine

Invariantul buclei for: este format din $\text{Perm}(u, v)$, unde u este valoarea curentă a variabilei a , și din proprietatea că primele $j-1$ sunt ordonate crescător:

$\text{Perm}(u, v) \wedge \text{SORTED}(a[0..j-1])$, unde u este valoarea curentă a variabilei a

Invariantul buclei while: elementele mutate $a[i+1..j-1]$ sunt mai mari decât $a[j]$ inițial ($= \text{temp}$): $a[i+1], \dots, a[j-1] > \text{temp}$.

Invariantul buclei **while** și condiția de terminare $a[i] \leq \text{temp} \vee i < 0$ asigură determinarea corectă a lui i , i.e. $\text{SORTED}(a[0..j])$.

Evaluarea algoritmului InsertSort 2/2

Timp de execuție

- dimensiune instanță: n ($= \mathbf{a.size()}$)
- operații măsurate: comparațiile care implică elementele tabloului
- cazul cel mai nefavorabil: când secvența de intrare este ordonată descrescător
 - căutarea poziției i în subsecvența $\mathbf{a}[0 .. j - 1]$ necesită $j - 1$ comparații
- numărul de comparații pentru acest caz este $1 + 2 + \dots + (n - 1) = \frac{(n - 1)n}{2} = O(n^2)$

Selecția sistematică

Analiza domeniului problemei Se bazează pe structura de date de tip *max-heap*. Proprietatea *MAXHEAP*(**a**):

$$\begin{aligned} (\forall i \geq 0) 2i + 1 < n &\implies a[i] \geq a[2i + 1] \wedge \\ 2(i + 1) < n &\implies a[i] \geq a[2(i + 1)] \end{aligned}$$

$$MAXHEAP(\mathbf{a}) \implies \max \mathbf{a} = \mathbf{a}[0]$$

Ideea algoritmului:

- se presupune *MAXHEAP*(**a**)
- dacă facem interschimbarea $\mathbf{swap}(\mathbf{a}, 0, \mathbf{n}-1)$, noua valoare $a[n - 1]$ e pe locul ei final și tabloul rămas de sortat este $\mathbf{a}[0..n - 2]$
- $\mathbf{a}[0..n - 2]$ se sortează în aceeași manieră

Ideea algoritmului mai algoritmică

```
heapSort(a, n) {  
    stabilește MAXHEAP( a)  
    for (r = n-1; r > 0; --r) {  
        swap(a, 0, r);  
        restabilește MAXHEAP( a[0..r - 1])  
    }  
}
```

(Acesta este pseudocod!)

Stabilirea proprietății de max-heap

Analiza domeniului problemei

- *MAXHEAP*(**a**, ℓ):

$$\begin{aligned} (\forall i \geq \ell) 2i + 1 < n &\implies a[i] \geq a[2i + 1] \wedge \\ 2(i + 1) < n &\implies a[i] \geq a[2(i + 1)] \end{aligned}$$

- $\ell \geq n/2 \implies \text{MAXHEAP}(\mathbf{a}, \ell)$
- dacă $\text{MAXHEAP}(\mathbf{a}, \ell-1)$ putem stabili $\text{MAXHEAP}(\mathbf{a}, \ell)$ inserând $a[\ell-1]$ în $\mathbf{a}[\ell..n-1]$

De la domeniul problemei la algoritmul:

```

j = ℓ;
while (există copil/copii a/ai lui j) {
    k = indexul copilului cu valoare maximă;
    if (a[j] < a[k]) swap(a, j, k);
    j = k;
}

```

Algoritmul HeapSort

```

insertInHeap(a, n, ℓ) {
    isHeap = false; j = ℓ;
    while (2*j+1 <= n-1 && ! isHeap) {
        k = 2*j +1;
        if ((k < n-1) && (a[k] < a[k+1])) k = k+1;
        if (a[j] < a[k]) swap(a, j, k); else isHeap = true;
        j = k;
    }
}

heapSort(a, n) {
    for (l = (n-1)/2; l >= 0; l = l-1)
        insertInHeap(a, n, l);
    r = n-1;
    while (r >= 1) {
        swap(a, 0, r);
        insertInHeap(a, r, 0);
        r = r - 1;
    }
}

```

Evaluarea algoritmului HeapSort 1/2

Corectitudine Se bazează pe corectitudinea implementării operațiilor peste *max-heap*. *invariantul instrucțiunii while din insertInHeap*: $(\forall i \geq \ell)$ dacă j nu este în arborele cu rădăcina în i , atunci $\text{MAXHEAP}(\mathbf{a}, i)$

invariantul lui for din heapSort: $\text{MAXHEAP}(\mathbf{a}, \ell)$

invariantul instrucțiunii while din heapSort: $\text{MAXHEAP}(\mathbf{a}[0..r-1]) \wedge \text{SORTED}(\mathbf{a}[r..n-1])$

Evaluarea algoritmului HeapSort 2/2

Țimp de execuție

- dimensiune instanță: n ($= \mathbf{a.size}()$)
- operații măsurate: comparațiile care implică elementele tabloului

- cazul cel mai nefavorabil: greu de spus
 - complexitatea timp al operației `insertInHeap`: $O(\log(n - \ell))$
 - dar construcția max-heap-ului necesită $O(n \log n) = O(\frac{n-1}{2}) + \dots + O(n)$ (se poate arăta că de fapt e $\Theta(n)$, a se vedea Cormen et al., 6.3)
 - complexitatea lui `while`: $O(\log(n-1)) + O(\log(n-2)) + \dots + O(\log 1) = O(n \log n)$
- numărul de comparații pentru acest caz este $O(n \log n)$

Alți algoritmi de sortare

Exerciții pentru seminar.

Două întrebări despre algoritmi de sortare

Algoritmii de sortare prezentați până acum se bazează pe executarea a două operații primitive: compararea și interschimbarea a două elemente. Deoarece orice interschimbare este, în general, precedată de o comparație (prin care se decide dacă interschimbarea este necesară) putem spune că operațiile de comparare domină calculul oricărui algoritm prezentat până acum.

Ne punem următoarele două întrebări:

- care este numărul minim de comparații executate în cazul cel mai nefavorabil?
- care algoritmi de sortare realizează minimumul de comparații, i.e. care algoritmi sunt optimali?

Pentru a putea răspunde la cele două întrebări trebuie mai întâi să precizăm modelul de calcul peste care sunt construiți acești algoritmi.

Arborii de decizie pentru sortare: intuitiv

Pentru simplitate vom presupune $a_i \neq a_j$ dacă $i \neq j$. Deoarece răspunsul dat de comparația $i ? j$ are numai două posibilități de alegere, rezultă că putem reprezenta cele două multi-multimi de comparații prin intermediul unui arbore binar:

- vârfurile interne conțin comparații $i ? j$;
- subarboarele din stânga conține comparațiile făcute în cazul $a_i < a_j$;
- subarboarele din dreapta conține comparațiile făcute în cazul $a_i > a_j$;
- vârfurile externe (frontiera) conțin permutări

Algoritmi reprezentați ca arborii de decizie (pentru sortare)

Definition

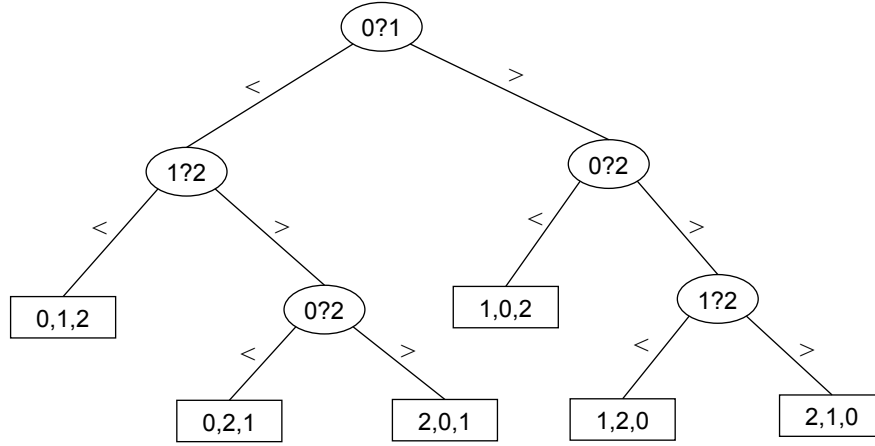
Definition

Arbori de decizie pentru sortare

Definition

Fie t un arbore de decizie pentru n elemente. Spunem că t **rezolvă problema sortării** dacă pentru orice intrare $a = (a_0, \dots, a_{n-1})$, calculul lui t pentru a se termină într-un vârf cu permutarea π astfel încât $a_{\pi(0)} < \dots < a_{\pi(n-1)}$. Un arbore de decizie care rezolvă problema sortării va mai fi numit și **arbore de decizie pentru sortare** iar modelul de calcul va fi numit **modelul arborilor de decizie pentru sortare**.

Arborele de decizie pentru InsertSort



Complexitatea sortării

Putem defini acum timpul de execuție minim pentru cazul cel mai nefavorabil prin expresia:

$$T(n) = \min_t \max_{\pi} \text{length}(\pi, t)$$

unde $\text{length}(\pi, t)$ reprezintă lungimea drumului de la rădăcină la vârful pe frontieră etichetat cu π în arborele de decizie pentru sortare t .

Theorem

Problema sortării are timpul de execuție pentru cazul cel mai nefavorabil $\Omega(n \log n)$ în modelul arborilor de decizie pentru sortare.

Demonstrație. Un arbore de decizie de dimensiune n care rezolvă problema sortării are $n!$ vârfuri pe frontieră. Un arbore de înălțime k are cel mult 2^k vârfuri pe frontieră. De aici rezultă

$$2^{T(n)} \geq n!$$

care implică $T(n) \geq \log_2(n!) = \Theta(n \log_2 n)$.

sfdem

Corollary

Algoritmul HeapSort este optimal în modelul arborilor de decizie pentru sortare.

3 Complexitatea căutării divide-et-impera

Problema căutării

Instance o mulțime univers \mathcal{U} , o submulțime $S \subseteq \mathcal{U}$ și un element a din \mathcal{U} ;

Question $a \in S$?

Presupunem că \mathcal{U} este total ordonată și mulțimea S este reprezentată de tabloul $s[0..n-1]$ cu $s[0] < \dots < s[n-1]$.

Algoritm generic divide-et-impera de căutare: ideea

Mai întâi generalizăm problema presupunând că se caută a în secvența $(s[p], \dots, s[q])$. Reamintim că are loc $s[p] < \dots < s[q]$. Algoritmii de căutare bazați pe paradigma divide-et-impera au o descriere recursivă definită după următoarea strategie:

- se determină m cu $p \leq m \leq q$;
- dacă $a = s[m]$ atunci căutarea se termină cu succes;
- dacă $a < s[m]$ atunci căutarea continuă cu subsecvența $(s[p], \dots, s[m-1])$;
- dacă $a > s[m]$ atunci căutarea continuă cu subsecvența $(s[m+1], \dots, s[q])$;

În funcție de modul de alegere a valorii m prin instrucțiunile 2 și 5, se disting mai mulți algoritmi de căutare. Cei mai cunoscuți dintre acestia sunt:

- **Căutare liniară** (secvențială). Se alege $m = p$.
- **Căutare binară**. Se alege $m = \lceil \frac{p+q}{2} \rceil$.
- **Căutare Fibonacci**. Se presupune $q+1-p = \text{Fib}(k) - 1$ unde $\text{Fib}(k)$ este la k -lea număr Fibonacci. Se alege m astfel încât $m-p = \text{Fib}(k-1)-1$ și $q-m = \text{Fib}(k-2)-1$.

Algoritm generic divide-et-impera de căutare

```
pos(s, n, a) {
  p = 0; q = n - 1;
  2: alege m între p și q
  while ( (a != s[m]) && (p < q)) {
    if (a < s[m]) q = m - 1; else p = m + 1;
    5: alege m între p și q
  }
  if (a == s[m]) return m; else return -1;
}
```

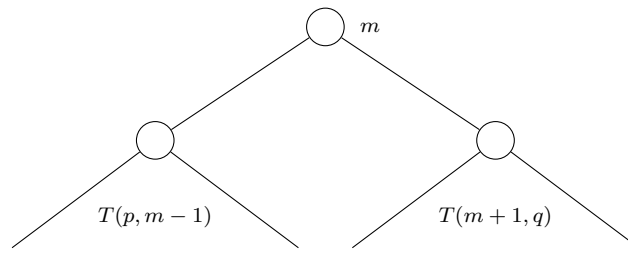
Algoritmi reprezentați ca arbori de decizie (pentru căutare)

Definition

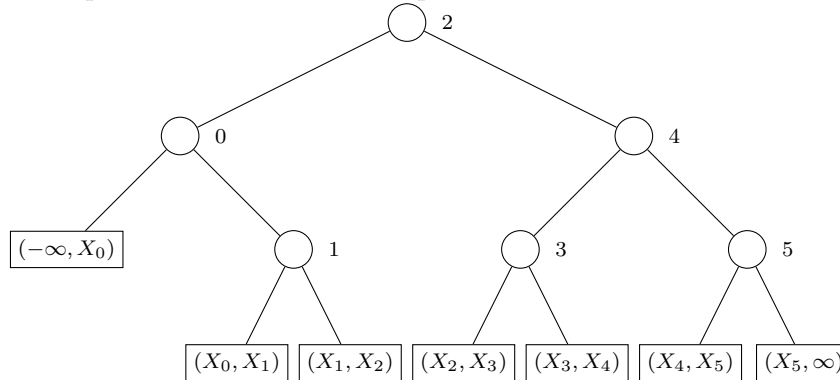
Arborele de decizie pentru căutare de dimensiune n atașat unui algoritm bazat pe metoda divide-et-impera este definit după cum urmează:

- Mai întâi se definește recursiv arborele $T(p, q)$ astfel:
 - dacă $p > q$ atunci $T(p, q)$ este arborele vid;
 - altfel, rădăcina este m calculat de instrucțiunea 2 sau 5, iar subarboarele stâng este $T(p, m - 1)$ și cel drept este $T(m + 1, q)$.
- Arborele de decizie pentru căutare de dimensiune n este $T(0, n - 1)$ la care se adaugă vârfurile externe având ca etichete intervalele $(-\infty, X_0), (X_0, X_1), \dots, (X_{n-1}, +\infty)$ în această ordine de la stânga la dreapta, unde X_0, \dots, X_{n-1} sunt n variabile.

$T(p, q)$ grafic



Exemplu de arbore de decizie pentru căutarea binară



Algoritmi reprezentați ca arbori de decizie (pentru căutare)

Definition

Calculul unui arbore de decizie pentru intrarea x_0, \dots, x_{n-1}, a , unde $x_0 < \dots < x_{n-1}$, constă în:

1. etichetarea nodurilor interne cu x_0, \dots, x_{n-1} astfel încât lista inordine ne dea ordinea crescătoare a etichetelor, și

2. parcurgerea unui drum de la rădăcină spre frontieră determinat astfel: dacă vârful curent v este etichetat cu x_m (reamintim că m este dat de instrucțiunea 2 sau 5 din schema procedurală divide-et-impera) atunci:
 - (a) dacă v este vârf extern etichetat cu (X_i, X_{i+1}) atunci $a \in (x_i, x_{i+1})$ (variabila X_i este interpretată ca având valoarea x_i) și calculul se **termină cu insucces**;
 - (b) dacă $a = x_m$ atunci calculul se **termină cu succes**;
 - (c) dacă $a < x_m$ atunci rădăcina subarborelui stâng devine vârf curent;
 - (d) dacă $a > x_m$ atunci rădăcina subarborelui drept devine vârf curent.

Cazul particular al căutării binare

Lemma

Fie t arborele de decizie pentru căutare cu n vârfuri corespunzător căutării binare. Dacă $2^{h-1} \leq n < 2^h$, atunci înălțimea lui t este h .

Demonstrație. Procedăm prin inducție după n . Dacă $n = 1$, atunci afirmația din lema este evident adevărată. Presupunem $n > 1$. Valoarea m corespunzătoare rădăcinii este $\lceil \frac{n}{2} \rceil$. Din definiția părții întregi superioare rezultă următoarele inegalități:

$$2^{h-2} \leq m < 2^{h-1} + 1. \quad (1)$$

Subarborii rădăcinii au m și respectiv $n - m - 1$ vârfuri astfel încât $m - 1 \leq n - m - 1 \leq m$. Dacă $m = n - m - 1$, atunci $m = \frac{n - 1}{2}$. Deoarece $n \leq 2^h - 1$, rezultă că $m \leq 2^{h-1} - 1 < 2^{h-1}$. Dacă $m - 1 = n - m - 1$, atunci $m = \frac{n}{2} < 2^{h-1}$. Rezultă $m < 2^{h-1}$ în toate cazurile. Aplicând ipoteza inductivă, rezultă că subarboarele cel mai înalt (cel cu m vârfuri și aflat la stânga rădăcinii) are înălțimea $h - 1$. Din definiția înălțimii arborelui binar, rezultă că înălțimea lui t este h . sfdem

Corollary

Timpul de execuție pentru cazul cel mai nefavorabil al căutării binare este $O(\log_2 n)$.

Proprietăți ale arborilor de decizie pentru căutare

Definition

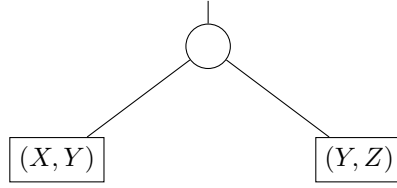
Fie t un arbore de decizie pentru căutare. **Lungimea internă** a lui t , notată $\text{IntLength}(t)$, este suma lungimilor drumurilor de la rădăcină la vârfurile interne. **Lungimea externă** a lui t , notată $\text{ExtLength}(t)$, este suma lungimilor drumurilor de la rădăcină la vârfurile de pe frontieră (pendante).

Lemma

Fie t un arbore de decizie pentru căutare cu n vârfuri interne. Atunci:

$$\text{ExtLength}(t) - \text{IntLength}(t) = 2n.$$

Demonstrație. Procedăm prin inducție după n . Pentru $n = 1$ avem $\text{IntLength}(t) = 0$ și $\text{ExtLength}(t) = 2$. Presupunem $n > 1$. Fie i un vârf intern cu ambii fii pe frontieră. Înlocuim subarboarele:



cu subarborele:



Noul arbore t' este un arbore de decizie cu $n - 1$ vârfuri interne și conform ipotezei inductive avem

$$\text{ExtLength}(t') - \text{IntLength}(t') = 2(n - 1).$$

Deoarece $\text{ExtLength}(t) = \text{ExtLength}(t') + k + 2$ și $\text{IntLength}(t) = \text{IntLength}(t') + k$, unde k este lungimea drumului de la rădăcină la i , rezultă:

$$\begin{aligned} \text{ExtLength}(t) - \text{IntLength}(t) &= \text{ExtLength}(t') + k + 2 - (\text{IntLength}(t') + k) \\ &= 2n - 2 + k + 2 - k \\ &= 2n. \end{aligned}$$

sfdem

Lemma

Lungimea internă minimă a unui arbore de decizie cu n vârfuri interne este:

$$(n + 1)(h - 1) - 2^h + 2$$

unde $h = \lceil \log_2(n + 1) \rceil$.

Demonstrație. Considerăm formula:

$$x + x^2 + \cdots + x^k = \frac{x^{k+1} - x}{x - 1}.$$

Derivăm:

$$1 + 2x + \cdots + kx^{k-1} = \frac{kx^{k+1} - (k+1)x^k + 1}{(x-1)^2}.$$

Înmulțim cu x :

$$x + 2x^2 + \cdots + kx^k = x \cdot \frac{kx^{k+1} - (k+1)x^k + 1}{(x-1)^2}.$$

Luăm $k = h - 1$ și $x = 2$:

$$2 + 2 \cdot 2^2 + \cdots + (h-1) \cdot 2^{h-1} = 2^h(h-2) + 2.$$

Lungimea internă minimă a unui arbore de decizie este suma primilor n termeni din seria:

$$0 + 1 + 1 + 2 + 2 + 2 + 2 + 3 + 3 + 3 + 3 + 3 + 3 + 3 + 3 + \cdots$$

și corespunde arborelui binar complet. Dacă presupunem $n + 1 = 2^h$ atunci această sumă este egală cu:

$$2 + 2 \cdot 2^2 + \dots + (h - 1)2^{h-1} = 2^h(h - 2) + 2.$$

Pentru cazul general trebuie să scădem suma drumurilor care unesc rădăcina cu vârfurile de pe nivelul h care lipsesc:

$$2^h(h - 2) + 2 - (2^h - 1 - n)(h - 1) = (n + 1)(h - 1) - 2^h + 2.$$

sfdem

Corollary

Lungimea externă minimă a unui arbore de decizie este

$$(n + 1)(h + 1) - 2^h.$$

Complexitatea căutării

Theorem

Problema căutării are timpul de execuție în cazul cel mai nefavorabil $\Omega(\log n)$ în modelul arborilor de decizie pentru căutare.

Demonstrație. Arborele binar cu lungimea internă minimă considerat în lema ?? are și înălțime minimă. Acum concluzia teoremei rezultă din faptul că $2^{h-1} \leq n < 2^h$, unde n este numărul de noduri iar h este înălțimea arborelui. sfdem

Corollary

Căutarea binară este optimă în modelul arborilor de decizie pentru căutare.

Demonstrație. Se observă că arborele de decizie asociat căutării binare are lungimea externă minimă. sfdem

4 Reducerea polinomială problemelor

Motivație

Mentalitate: "Dacă știu să rezolv problema Q , pot utiliza acel algoritm să rezolv P ?"

Intuitiv: Problema P se reduce la Q dacă un algoritm care rezolvă Q poate ajuta la rezolvarea lui P .

Aplicații:

- proiectarea de algoritmi
- demonstrarea limitelor: dacă P este dificilă atunci și Q este dificilă
- clasificarea problemelor

Reducerea Turing/Cook

Problema P se reduce polinomial la problema (rezolvabilă) Q , notăm $P \propto Q$, dacă se poate construi un algoritm care rezolvă P după următoarea schemă:

1. se consideră la intrare o instanță p a lui P ;
2. preprocesează în timp polinomial intrarea p
3. se apelează algoritmul pentru Q , posibil de mai multe ori (un număr polinomial)
4. se postprocesează rezultatul dat de Q în timp polinomial

Dacă pașii de preprocesare și postprocesare necesită $O(g(n))$ timp, atunci scriem $P \propto_{g(n)} Q$.

Exemplu: $\text{MAX} \propto \text{SORT}$

Fie MAX problema determinării elementului maxim dintr-o mulțime:

Input O mulțime S total ordonată.
Output Cel mai mare element din S .

Următorul algoritm rezolvă MAX :

1. reprezintă S cu un tablou s (preprocesare);
2. apelează un algoritm de sortare pentru s ;
3. întoarce ultimul element din s (postprocesarea);

Deoarece algoritmul de mai sus este mai complex decât algoritmul care determină maximum enumerând toate elementele din S , rezultă că \propto nu este întotdeauna o "reducere de la o problemă mai complexă la una mai simplă". De aceea termenul de "*transformare*" este mai potrivit. Menținem totuși și termenul de reducere pentru că așa este cunoscut în literatură.

Variante pentru submulțimea de sumă dată

SSD1

Input O mulțime S de numere întregi, M număr întreg pozitiv.
Output Cel mai mare număr întreg M^* cu proprietățile $M^* \leq M$ și există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M^*$.

SSD2

Instance O mulțime S de numere întregi, M, K două numere întregi pozitive cu $K \leq M$.
Question Există număr întreg M^o cu proprietățile $K \leq M^o \leq M$ și $\sum_{x \in S'} x = M^o$ pentru o submulțime oarecare $S' \subseteq S$?

SSD3

Instance O mulțime S de numere întregi, M un număr întreg pozitiv.
Question Există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M$?

Exemplu: SSD1 \propto SSD2*SSD1**Input* O mulțime S de numere întregi, M număr întreg pozitiv.*Output* Cel mai mare număr întreg M^* cu proprietățile $M^* \leq M$ și există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M^*$.*SSD2**Instance* O mulțime S de numere întregi, M, K două numere întregi pozitive cu $K \leq M$.*Question* Există număr întreg M° cu proprietățile $K \leq M^\circ \leq M$ și $\sum_{x \in S'} x = M^\circ$ pentru o o submulțime oarecare $S' \subseteq S$?

1. nu există preprocesare;
2. caută binar pe M^* în intervalul $(0, M]$ apelând un algoritm care rezolvă SSD2;

Acesta este un exemplu de reducerea unei probleme de optim la versiunea ei ca problemă de decizie.

Exemplu: SSD2 \propto SSD1*SSD1**Input* O mulțime S de numere întregi, M număr întreg pozitiv.*Output* Cel mai mare număr întreg M^* cu proprietățile $M^* \leq M$ și există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M^*$.*SSD2**Instance* O mulțime S de numere întregi, M, K două numere întregi pozitive cu $K < M$.*Question* Există număr întreg M° cu proprietățile $K \leq M^\circ \leq M$ și $\sum_{x \in S'} x = M^\circ$ pentru o o submulțime oarecare $S' \subseteq S$?

1. nu există preprocesare;
2. calculează $M^* \leq M$ apelând un algoritm care rezolvă SSD1;
3. dacă $M^* \geq K$ întoarce 'DA', altfel întoarce 'NU';

Exemplu: SSD3 \propto SSD1*SSD1**Input* O mulțime S de numere întregi, M număr întreg pozitiv.*Output* Cel mai mare număr întreg M^* cu proprietățile $M^* \leq M$ și există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M^*$.*SSD3**Instance* O mulțime S de numere întregi, M un număr întreg pozitiv.*Question* Există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M$?

1. nu există preprocesare;
2. calculează $M^* \leq M$ apelând un algoritm care rezolvă SSD1;
3. dacă $M^* = M$ întoarce 'DA', altfel întoarce 'NU';

Reducerea Karp

Se consideră P și Q probleme de decizie.

Problema P se reduce polinomial la problema (rezolvabilă) Q , notăm $P \propto Q$, dacă se poate construi un algoritm care rezolvă P după următoarea schemă

1. se consideră la intrare o instanță p a lui P ;
2. preprocesează în timp polinomial intrarea p
3. se apelează (o singură dată) algoritmul pentru Q
4. răspunsul pentru Q este același cu cel al lui P (fără postprocesare)

Dacă pasul de preprocesare necesită $O(g(n))$ timp, atunci scriem $P \propto_{g(n)} Q$.

Reducerea Karp este un caz particular de reducere Turing/Cook.

Exemplu: SSD3 \propto SSD2

SSD2

Instance O mulțime S de numere întregi, M, K două numere întregi pozitive cu $K \leq M$.

Question Există număr întreg M° cu proprietățile $K \leq M^\circ \leq M$ și $\sum_{x \in S'} x = M^\circ$ pentru o o submulțime oarecare $S' \subseteq S$?

SSD3

Instance O mulțime S de numere întregi, M un număr întreg pozitiv.

Question Există o submulțime $S' \subseteq S$ cu $\sum_{x \in S'} x = M$?

1. nu există preprocesare;
2. apelează un algoritm care rezolvă SSD2 pentru instanța S, M, M ;

Exemplu: SUBSET \propto DISJOINT

SUBSET

Instanță Două mulțimi S_1 și S_2 ($S_1, S_2 \subseteq \mathcal{U}$, \mathcal{U} mulțime univers).

Întrebare $S_1 \subseteq S_2$?

DISJOINT

Instanță Două mulțimi S_1 și S_2 .

Întrebare $S_1 \cap S_2 = \emptyset$?

SUBSET \propto DISJOINT:

1. se consideră la intrare o instanță S_1, S_2 a lui SUBSET;
2. calculează $t(S_1, S_2) = S_1, \overline{S_2}$
3. întoarce rezultatul întors de un algoritm care rezolvă DISJOINT pentru instanța $S_1, \overline{S_2}$.

Reducerea: proprietăți

Theorem

a) Dacă P are complexitatea timp $\Omega(f(n))$ și $P \propto_{g(n)} Q$ (versiunea Karp) atunci Q are complexitatea timp $\Omega(f(n) - g(n))$. [2ex] b) Dacă Q are complexitatea $O(f(n))$ și $P \propto_{g(n)} Q$ (versiunea Karp) atunci P are complexitatea $O(f(n) + g(n))$.