Proiectarea Algoritmilor

I CB: Stefan Trausan-Matu stefan.trausan@cs.pub.ro

I CC: Costin Chiru costin.chiru@cs.pub.ro

I CA: Traian Rebedea traian.rebedea@cs.pub.ro

Obiectivele cursului (I)

- Cunoasterea unui set important de algoritmi si metode de rezolvare a problemelor de algoritmica
- Dezvoltarea abilitatilor de adaptare a unui algoritm la o problema din viata reala
- Dezvoltarea abilitatilor de lucru in echipa

Obiectivele cursului (II)

- Utilizarea teoriei predate la curs pentru proiectarea algoritmilor de rezolvare pentru probleme tipice întâlnite în practica dezvoltării sistemelor de programe.
- Discutarea relaţiei dintre caracteristicile problemelor, modul de rezolvare şi calitatea soluţiilor.
- Compararea variantelor unor algoritmi pentru rezolvarea problemelor dificile.

De ce sa invat PA?

- Exemple de utilizari ale PA-ului in diferite meserii:
  - web developer – web social, teoria grafurilor, data mining, clustering;
  - game dev – cautari, grafuri, inteligenta artificiala,
  - project manager – fluxuri, grafuri de activitati,
  - dezvoltator de sisteme de operare – structuri de date avansate, scheme de algoritmi
  - programator – tot ce tine de algoritmi, in special complexitate si eficienta
  - tester – tot ce tine de algoritmi, in special complexitate, eficienta si debugging

Planul cursului (I)

- Scheme de algoritmi
  - Caracteristici ale problemelor şi tehnici asociate de rezolvare: divide&impera, rezolvare lacomă (arbori Hufmann), programare dinamică (AOC). Backtracking cu optimizări. Propagarea restricţiilor.
- Algoritmi pentru jocuri – minimax şi α-β.
- Algoritmi pentru grafuri
  - Algoritmi pentru grafuri: parcurgeri, sortare topologică, componente tare conexe, articulaţii, punţi, arbori minimi de acoperire, drumuri de cost minim, fluxuri.

Planul cursului (II)

- Rezolvarea problemelor prin căutare euristică
  - Rezolvarea problemelor prin căutare euristică A*. Completitudine şi optimalitate, caracteristici ale euristicilor.
- Algoritmi aleatorii
  - Algoritmi aleatorii. Las Vegas şi Monte Carlo, aproximare probabilistică.

Evaluarea

- Cititi documentul REGULAMENT PA 2010 de pe site!
- Examen 4 p
- Laborator 6 p ☺
  - 3p teme (4 teme punctate egal)
  - 2p laborator
  - 2p proiect
- Activitate stiintifica – maxim 0,5p bonus
- Obs. 1 - Nu se copiaza in facultate!
- Obs. 2 - Prima tema copiata se puncteaza cu minus valoarea maxima a temei. La a doua tema copiata, se repeta materia.

Proiect PA (1)

- Realizarea unui joc de sah in echipe de cate 4 studenti.
- La finalul proiectului se va face un concurs intre echipele participante – la nivelul grupei, seriei si anului.
- Obiective
  - Rezolvarea unei probleme interesante din viaţa reală
  - Aplicarea unor algoritmi din domeniul jocurilor
  - Cunoasterea si utilizarea unor structuri de date avansate
  - Dezvoltarea abilitatilor de lucru in echipa
  - Dezvoltarea spiritului competitiv

Proiect PA (2)

- Etape:
  - 0. Formarea echipelor
  - 1. Documentare + reprezentarea datelor + comunicaţie cu WinBoard
  - 2. Mutări complete + interpretare mutări adversar
  - 3. Minimax + bază de date deschideri/închideri
  - 4. Îmbunătăţire algoritm de joc
- Fiecare din etapele 1-4 presupune realizarea unei aplicaţii funcţionale
- După predarea etapei 4 va avea loc concursul la nivel de serie

Feedback 2008 + 2009

- Idei preluate din feedback 2008:
  - Schimbarea modului de organizare al proiectului (etape, mod de lucru in echipă)
  - Schimbarea orientării temelor (variante mai usoare, notare parţială)
  - Subliniată importanţa bibliografiei
- Idei preluate din feedback 2009:
  - Eliminare restricţii echipa proiect la nivel de grupă
  - Publicare teme pe infoarena
  - Fără net în laboratoare
  - Laboratoare mai bune, teme mai atent elaborate, organizare mai bună (sperăm să ne iasă ☺)

Bibliografie

- Introducere in Analiza Algoritmilor de *Cristian Giumale* – Ed. Polirom 2004
- Introducere in Algoritmi *de Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest* – Ed. Agora
- Vedeţi şi recomandările din Regulament!
- Mentiune importanta – **slide-urile reprezinta doar un suport pentru prezentare**

Proiectarea Algoritmilor

Curs 1 – Scheme de algoritmi – Divide et impera + Greedy

Curs 1 – Cuprins

- Scheme de algoritmi
- Divide et impera
- Exemplificare folosind Merge sort
- Alte exemple de algoritmi divide et impera
- Greedy
- Exemplificare folosind arbori Huffman
- Demonstratia corectitudinii algoritmului Huffman

Curs 1 – Bibliografie

- Giumale – Introducere in Analiza Algoritmilor cap 4.4
- Cormen – Introducere în Algoritmi cap. 17
- http://www.cs.ucsb.edu/~suri/cs235/ClosestPair.pdf
- http://www.cs.umass.edu/~barring/cs611/lecture/4.pdf
- http://thor.info.uaic.ro/~dlucanu/cursuri/tpaa/resurse/Curs6.pps
- http://www.math.fau.edu/locke/Greedy.htm
- http://en.wikipedia.org/wiki/Greedoid

Scheme de algoritmi

- Prin scheme de algoritmi înțelegem tipare comune pe care le putem aplica in rezolvarea unor probleme similare.
- O gama larga de probleme se poate rezolva folosind un număr relativ mic de scheme
- => Cunoașterea schemelor determina o rezolvare mai rapida si mai eficienta a problemelor

Divide et impera (I)

- Ideea (divide si cucerește) este atribuita lui Filip al II-lea, regele Macedoniei (382-336 i.e.n.), tatăl lui Alexandru cel Mare si se refera la politica acestuia fata de statele grecești.
- In CS – Divide et impera se refera la o clasa de algoritmi care au ca principale caracteristici faptul ca împart problema in subprobleme similare cu problema inițiala dar mai mici ca dimensiune, rezolva problemele recursiv si apoi combina soluțiile pentru a crea o soluție pentru problema originala.

Divide et impera (II)

- Schema Divide et impera consta in 3 pași la fiecare nivel al recurentei:
  - Divide problema data intr-un număr de subprobleme
  - Impera (cucerește) – subproblemele sunt rezolvate recursiv. Daca subproblemele sunt suficient de mici ca date de intrare se rezolva direct (ieșirea din recurenta)
  - Combina – soluțiile subproblemelor sunt combinate pentru a obține soluția problemei inițiale

Divide et impera – Avantaje si Dezavantaje

- Avantaje:
  - Produce algoritmi eficienti
  - Descompunerea problemei in subprobleme faciliteaza paralelizarea algoritmului in vederea executiei sale pe mai multe procesoare
- Dezavantaje:
  - Se adauga un overhead datorat recursivitatii (retinerea pe stiva a apelurilor functiilor)

Merge sort (I)

- Algoritmul Merge Sort este un exemplu clasic de rezolvare cu D&I
- Divide: Divide cele n elemente ce trebuie sortate in 2 secvențe de lungime n/2
- Impera: Sortează secvențele recursiv folosind *merge sort*
- Combina: Secvențele sortate sunt asamblate pentru a obține vectorul sortat
- Recurența se oprește când secvența ce trebuie sortata are lungimea 1 (un vector cu un singur element este întotdeauna sortat ☺)
- Operația cheie este asamblarea soluțiilor parțiale.

Merge Sort (II)

- Algoritm [Cormen]
  - MERGE-SORT(A, p, r)
  - 1 **if** $p < r$
  - 2 **then** $q \leftarrow [(p + r)/2]$ // divide
  - 3 MERGE-SORT(A, p, q) //impera
  - 4 MERGE-SORT(A, $q + 1$, r)
  - 5 MERGE(A, p, q, r) // combina
    // (interclasare)

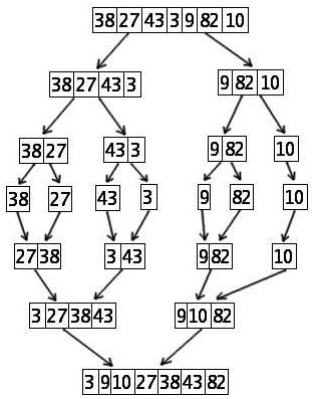
Merge Sort (III) – Algoritmul de interclasare

- Algoritm [Cormen]

```
MERGE(A, p, q, r) // p si r sunt capetele intervalului, q este "mijlocul"
1     n1 ← q - p + 1 // numarul de elemente din partea stanga
2     n2 ← r – q  // numarul de elemente din partea dreapta
3     create arrays L[1 -> n1 + 1] and R[1 -> n2 + 1]
4      for i ← 1 to n1
5           do L[i] ← A[p + i - 1] // se copiaza partea stanga in L
6     for j ← 1 to n2
7           do R[j] ← A[q + j] // si partea dreapta in R
8     L[n1 + 1] ← ∞
9     R[n2 + 1] ← ∞
10    i ← 1
11    j ← 1
12    for k ← p to r                    // se copiaza inapoi in vectorul de
13          do if L[i] ≤ R[j]           // sortat elementul mai mic din cei
14                then A[k] ← L[i]      // doi vectori sortati deja
15                i ← i + 1
16          else A[k] ← R[j]
17                j ← j + 1
```

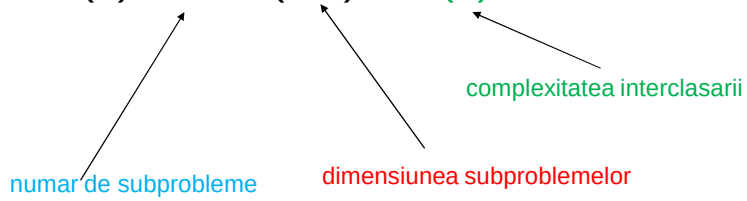

Exemplu functionare Merge Sort

- Exemplu functionare [Wikipedia]



MergeSort - Complexitate

- $T(n) = 2 * T(n/2) + \Theta(n)$



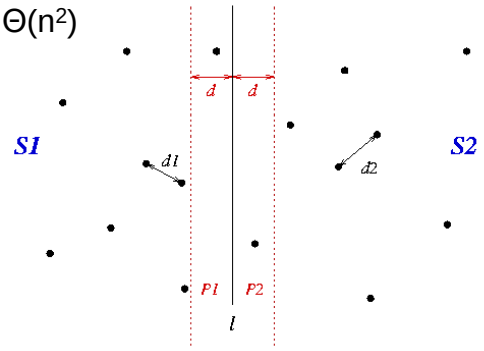
=> (din T. Master) $T(n) = \Theta(n * \log n)$

Divide et impera – alte exemple (I)

- Calculul puterii unui numar: x^n
  - Algoritm “clasic”
    - pentru i = 1 → n rez = rez * x; return rez
  - Complexitate: $\Theta(n)$
  - Algoritm divide et impera
    - daca n este par return $x^{n/2} * x^{n/2}$
    - altfel (n este impar) return $x * x^{(n-1)/2} * x^{(n-1)/2}$
  - Complexitate: $T(n) = T(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(\log n)$

Divide et impera – alte exemple (II)

- Calculul celei mai scurte distante intre 2 puncte din plan (http://www.cs.ucsb.edu/~suri/cs235/ClosestPair.pdf)
  - algoritmul naiv – $\Theta(n^2)$



Divide et impera – alte exemple (III)

- sortează punctele in ordinea crescătoare a coordonatei x ($\Theta(n \log n)$)
- împărțim setul de puncte in 2 seturi de dimensiune egala si calculam recursiv distanta minima in fiecare set (l = linia ce împarte cele 2 seturi, d = distanta minima calculata in cele 2 seturi)
- elimina punctele care sunt plasate la distanta de l > d
- sortează punctele ramase după coordonata y
- calculează distantele de la fiecare punct rămas la cei 5 vecini (nu pot fi mai multi)
- daca găsește o distanta < d, atunci actualizează d

Divide et impera – tema de gandire

- Se da o multime M de numere intregi si un numar x. Se cere sa se determine daca exista a,b∈ M a.i. a + b = x
- Algoritmul propus trebuie sa aiba complexitatea θ(nlogn)
- Temele de la curs sunt facultative☺

Greedy (I)

- Metoda de rezolvare eficienta a unor probleme de optimizare
- Soluția trebuie sa satisfacă un criteriu de optim global (greu de verificat) → optim local mai usor de verificat
- Se aleg soluții parțiale ce sunt îmbunătățite repetat pe baza criteriului de optim local pana ce se obțin soluții finale
- Solutiile partiale ce nu pot fi imbunătăţite sunt abandonate → proces de rezolvare irevocabil (fără reveniri)

Greedy (II)

- Schema generala de rezolvare a unei probleme folosind Greedy (programarea lacoma):

- Rezolvare_lacoma(Crit_optim, Problema)
  - 1. sol_partiale = sol_initiale(problema); // determinarea solutiilor partiale
  - 2. sol_fin = Φ;
  - 3. while (sol_partiale ≠ Φ)
  - 4. for-each(s in sol_partiale)
  - 5. if(s este o solutie a problemei) { // daca e solutie
  - 6. sol_fin = sol_fin U {s}; // finala se salveaza
  - 7. sol_partiale = sol_partiale \ {s};
  - 8. } else // se poate optimiza?
  - 9. if(optimizare_posibila(s, Crit_optim, Problema)) // da
  - 10. sol_partiale = sol_partiale \ {s} U optimizare(s,Crit_optim,Problema)
  - 11. else sol_partiale = sol_partiale \ {s}; // nu
  - 12. return sol_fin;

Arbori Huffman

- Metoda de codificare folosita la compresia fișierelor
- Construcția unui astfel de arbore se realizează printr-un algoritm greedy
- Consideram un text, de exemplu:
  - **ana are mere**
- Vom exemplifica pas cu pas constructia arborelui de codificare pentru acest text si vom defini pe parcurs conceptele de care avem nevoie.

Arbori Huffman – Definitii (I)

- K – mulțimea de simboluri ce vor fi codificate
- Arbore de codificare a cheilor K este un arbore binar ordonat cu proprietățile:
  - Doar frunzele conțin cheile din K; nu exista mai mult de o cheie intr-o frunză
  - Toate nodurile interne au exact 2 copii
  - Arcele sunt codificate cu 0 si 1 (arcul din stânga unui nod – codificat cu 0)
- k = Codul unei chei – este șirul etichetelor de pe calea de la rădăcina arborelui la frunza care conține cheia k (k este din K).
- p(k) – frecvența de apariție a cheii k in textul ce trebuie comprimat.
- Ex pentru “ana are mere”:
  - p(a) = p(e) = 0.25; p(n) = p(m) = 0.083;p(r) = p( ) = 0.166

Arbori Huffman – Definitii (II)

- A – arborele de codificare a cheilor
- lg_cod(k) – lungimea codului cheii k conform A
- nivel(k,A) – nivelul pe care apare in A frunza ce conține cheia K
- Costul unui arbore de codificare A al unor chei K relativ la o frecventa p este:

$$Cost(A) = \sum_{k \in K} lg\_cod(k) * p(k) = \sum_{k \in K} nivel(k, A) * p(k)$$

- Un arbore de codificare cu cost minim al unor chei K, relativ la o frecventa p este un arbore Huffman, iar codurile cheilor sunt coduri Huffman.

Arbori Huffman – algoritm de constructie (I)

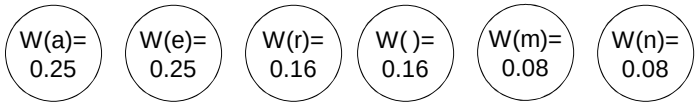
- 1. pentru fiecare k din K se construiește un arbore cu un singur nod care conține cheia k si este caracterizat de ponderea w = p(k). Subarborii construiți formează o mulțime numita Arb.
- 2. Se aleg doi subarbori a şi b din Arb astfel incât a şi b au pondere minima.

Arbori Huffman – algoritm de constructie (II)

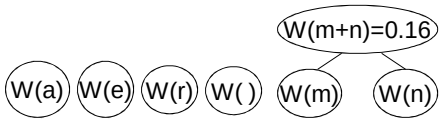
- 3. Se construieste un arbore binar cu o radacina *r* care nu contine nici o cheie si cu descendentii a si b. Ponderea arborelui este definita ca w(r) = w(a) + w(b)
- 4. Arborii a si b sunt eliminati din Arb iar r este inserat in Arb.
- 5. Se repeta procesul de constructie descris de pasii 2-4 pana cand multimea Arb contine un singur arbore – Arborele Huffman pentru cheile K

Arbori Huffman – Exemplu

- Text: ana are mere
- p(a) = p(e) = 0.25; p(n) = p(m) = 0.083; p(r) = p( ) = 0.166
- Pasul 1

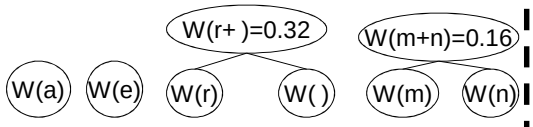


- Pasii 2-4

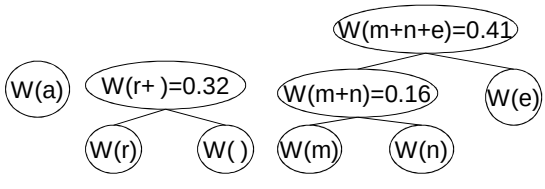


Arbori Huffman – Exemplu(II)

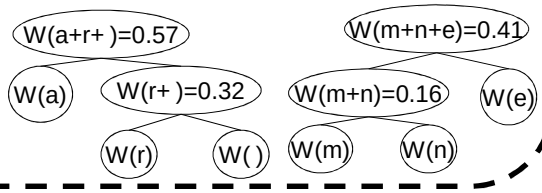
- Pasii 2-4 (II)



- Pasii 2-4 (III)

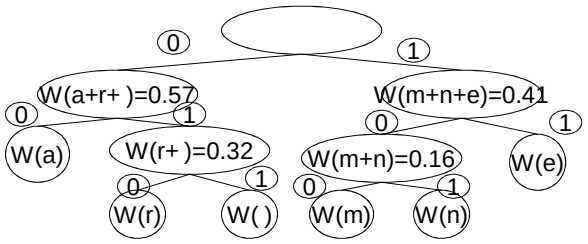


- Pasii 2-4 (IV)



Arbori Huffman – Exemplu (III)

- Pasii 2-4 (V)



- Codificare: a - 00; e -11; r - 010; ’ ‘ - 011; m - 100; n - 101;
- Cost(A) = 2 * 0.25 + 2 * 0.25 + 3 * 0.083 + 3 * 0.083 + 3 * 0.166 + 3 * 0.166 = 1 + 1.2 = 2.2 biti.

Arbori Huffman - pseudocod

- Huffman(K,p){
  - 1. Arb = {k ∈ K | frunza(k, p(k))};
  - 2. while (card(Arb) > 1)
  - 3. fie a_1 si a_2 arbori din Arb a.i. ∀a ∈ Arb a ≠ a_1 si a ≠ a_2, avem $w(a_1) \le w(a) \wedge w(a_2) \le w(a))$; // practic se extrage // de doua ori minimul si se salveaza in a_1 si a_2
  - 4. Arb = Arb \ {a_1, a_2} U nod_intern(a_1, a_2, $w(a_1) + w(a_2)$);
  - 5. if(Arb = Φ)
  - 6. return arb_vid;
  - 6. else
  - 7. fie A singurul arbore din multimea Arb;
  - 8. return A;
- Notatii folosite:
  - a = frunza(k, p(k)) – subarbore cu un singur nod care contine cheia k, iar w(a) = p(k);
  - a = nod_intern(a_1, a_2, x) – subarbore format dintr-un nod intern cu descendentii a_1 si a_2 si w(a) = x

Arbori Huffman - Decodificare

- Se incarca arborele si se decodifica textul din fisier conform algoritmului:

- Decodificare (in, out)

```
A = restaurare_arbore (in)      // reconstruiesc arborele
while(! terminare_cod(in))      // mai am caractere de citit
    nod = A                     // pornesc din radacina
    while (! frunza(nod))       // cat timp nu am determinat caracterul
        if (bit(in) = 1) nod = dreapta(nod)      // avansez in arbore
            else nod = stanga(nod)
    write(out, cheie(nod))      // am determinat caracterul si il scriu
```


Demonstratie (I)

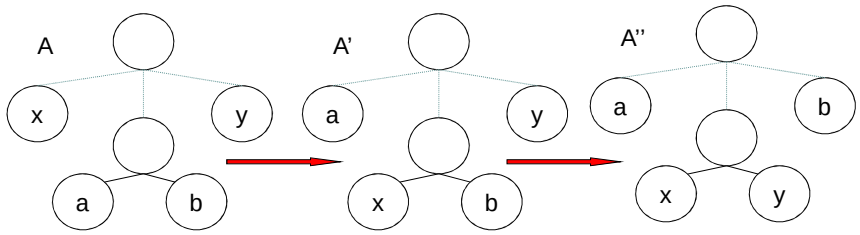
- Arborele de codificare construit trebuie să aibă cost minim pentru a fi arbore Huffman

- Lema 1. Fie K mulțimea cheilor dintr-un arbore de codificare, card(K) ≥ 2, x, y două chei cu pondere minimă. ∃ un arbore Huffman de înălțime h in care cheile x şi y apar pe nivelul h fiind descendente ale aceluiași nod intern.

Demonstratie (II)

- Demonstratie Lema 1:

$$Cost(A) = \sum_{k \in K} lg\_cod(k) * p(k) = \sum_{k \in K} nivel(k, A) * p(k)$$



- Se interschimbă a cu x şi b cu y şi din definiţia costului arborelui => cost(A'') ≤ cost(A') ≤ cost(A) => A'' arbore Huffman

Demonstratie (III)

- Lema 2. Fie A un arbore Huffman cu cheile K, iar x şi y două chei direct descendente ale aceluiaşi nod intern *a*. Fie K' = K \ {x,y} U {z} unde z este o cheie fictivă cu ponderea w(z) = w(x) + w(y). Atunci arborele A' rezultat din A prin inlocuirea subarborelui cu rădăcina *a* si frunzele x, y cu subarborele cu un singur nod care conţine frunza z, este un arbore Huffman cu cheile K'.

- Demonstratie
  - 1) analog Cost(A') ≤ Cost(A) (Cost(A) = Cost(A') + w(x) + w(y))
  - 2) pp există A'' a.i. Cost(A'') < Cost(A') =>
    - Cost(A'') < Cost(A) - (w(x) + w(y));
    - Cost(A'') + w(x) + w(y) < Cost(A); => A nu este Huffman (contradicţie)

Demonstratie (IV)

- Teoremă – Algoritmul Huffman construiește un arbore Huffman.
- Demonstrație prin inducție după numărul de chei din mulțimea K.
- n ≤ 2 => evident
- n > 2
  - Ip. Inductivă: algoritmul Huffman construiește arbori Huffman pentru orice mulțime cu n-1 chei
  - Fie K = $\{k_1, k_2, \dots, k_n\}$ a.i. $w(k_1) \le w(k_2) \le \dots \le w(k_n)$

Demonstratie (V)

- Cf. Lema 1, ∃ Un arbore Huffman unde cheile k_1, k_2 sunt pe același nivel şi descendente ale aceluiași nod.
- A_{n-1} – arborele cu n-1 chei K' = K - $\{k_1,k_2\} \cup z$ unde $w(z) = w(k_1) + w(k_2)$
- A_{n-1} rezultă din A_n prin modificările prezentate in Lema 2 => A_{n-1} este Huffman, şi cf. ipotezei inductive e construit prin algoritmul Huffman(K',p')
- => Algoritmul Huffman(K, p) construiește arborele format din k_1 si k_2 si apoi lucrează ca şi algoritmul Huffman(K', p') ce construiește A_{n-1} => construieste arborele Huffman(K, p)

Întrebări?