Tehnici de programare - TP

Cursul 13 – Analiza algoritmilor. Biblioteci software.

Ș.l. dr. ing. Cătălin Iapă

catalin.iapa@cs.upt.ro

De data trecută: Backtraking

Eficienta algoritmilor

Timpul de executie

Generarea de numere aleatoare

Biblioteci software

Metoda Backtracking - principii

Scopul algoritmului concret poate să fie determinarea unei soluţii rezultat sau a tuturor soluţiilor rezultat, fie în scopul afişării lor, fie pentru a alege una optimă din punctul de vedere al unor criterii de optimizare (minimizare sau maximizare).

O metodă simplă de selectare a soluţiilor rezultat este aceea de a genera toate soluţiile posibile şi de a verifica satisfacerea condiţiilor interne *(căutare exhaustivă* în întregul spațiu al soluțiilor posibile*).* Această metodă necesită însă un timp de execuţie foarte mare.

Metoda Backtracking - algoritm

Mai simplu spus, algoritmii de tip backtracking funcționează în felul următor:

- soluția problemei se construiește succesiv, pas cu pas
- dacă la un pas există mai multe posibilități de continuare, se vor încerca pe rând fiecare dintre ele
- dacă s-a ajuns într-un punct în care nu se mai poate continua, se revine la pasul anterior pentru a se încerca următoarea variantă posibilă
- după ce s-au epuizat toate posibilitățile de la pasul anterior, se revine cu încă un pas mai înainte și tot așa, în mod recursiv, până când au fost încercate toate posibilitățile din toți pașii

Metoda Backtracking – implementare

O implementare recursivă simplă a algoritmului backtracking în limbajul de programare C:

```
void back(int k){
 for(int i=0;i<n;i++)
      {
       v[k]=i;
       if (valid(k))
            if(solutie(k))
                 afisare();
            else
                  back(k+1);
        }
}
```

Metoda Backtracking – exerciții

Exercițiul 1:

Se primește un cuvânt din lina de comandă. Să se afișeze toate anagramele sale.(DEX anagramă: schimbare a ordinii literelor unui cuvânt, pentru a obține alt cuvânt; cuvânt obținut prin această schimbare.)

Metoda Backtracking – exerciții

```
int main(int argc, char* argv[])
{
char *cuv=strdup(argv[1]);
int n=strlen(cuv);
back(1, n, cuv);
return 0;
}
```

Metoda Backtracking – exerciții

```
void back(int k, int n, char* cuv)
{
for (int i=1; i<=n; i++){
        st[k]=i;
        if (valid(st, k)){
                if (solutie(st, k, n)){
                        afisare(st, k, cuv);        }
                else{
                        back(k+1, n, cuv);          }
                                }
        }
}
```

Metoda Backtracking – exerciții

```
int valid(int st[], int k){
        for (int i=1; i<k; i++){
                if (st[i]==st[k]){
                        return 0;       }
                }
        return 1;}

int solutie(int st[], int k, int n){
        return (k==n);}

void afisare(int st[], int k, char *cuv){
        for (int i=1; i<=k; i++){
                printf("%c", cuv[st[i]-1]);    }
        printf("\n");}
```

Metoda Backtracking – exerciții

Exercițiul 2:

Să se genereze toate șirurile de cifre distincte a căror sumă este egală cu n citit de la tastatură.

Metoda Backtracking – exerciții

```
int main(){
scanf("%d", &x);
back(0);
return 0;
}
```

Metoda Backtracking – exerciții

```
void back(int k){
for (int i = 1; i < n; i++)    {
        v[k] = i;
        if (valid(k))       {
                if (solutie(k))          {
                        afisare(k);          }
                else          {
                        back(k + 1);          }
                }
        }
}
```

Metoda Backtracking – exerciții

```
int valid(int k){
int suma = 0;
for (int i = 0; i < k; i++)    {
        if (v[i] == v[k])       {
                return 0;       }
        suma = suma + v[i];
        }
suma = suma + v[k];
        if (suma > x)    {
                return 0;    }
        else    {
                return 1;    }
}
```

Metoda Backtracking – exerciții

```
int solutie(int k){
int suma = 0;
for (int i = 0; i <= k; i++)    {
            suma = suma + v[i];    }
if (x == suma)    {
            return 1;    }
else    {
            return 0;    }
}
void afisare(int k){
for (int i = 0; i <= k; i++)    {
            printf("%d ", v[i]);    }
printf("\n");
}
```

Metoda Backtracking – exerciții

Problema celor 8 regine (Eight Queens)

Se cere să se realizeze programul care să plaseze opt regine pe o tablă de şah, astfel încât nici una dintre ele să nu le ameninţe pe celelalte. La jocul de şah, o regină “ameninţă” pe linii, coloane şi diagonale, pe orice distanţă.

Metoda Backtracking – exerciții

Problema celor 8 regine (Eight Queens)

Această problemă a fost investigată de Carl Friedrich Gauss în 1850 (care însă nu a rezolvat-o complet). Nici până în prezent problema nu are o soluţie analitică satisfăcătoare. În schimb ea poate fi rezolvată prin încercări, necesitând o mare cantitate de muncă, răbdare şi acurateţe (condiţii în care calculatorul se descurcă excelent).

Problema are 92 de soluţii din care, din motive de simetrie a tablei de șah, doar 12 sunt diferite.

Problema poate fi uşor extinsă pentru *n* regine plasate pe o tablă pătrată cu *n* linii şi *n* coloane.

Metoda Backtracking – exerciții

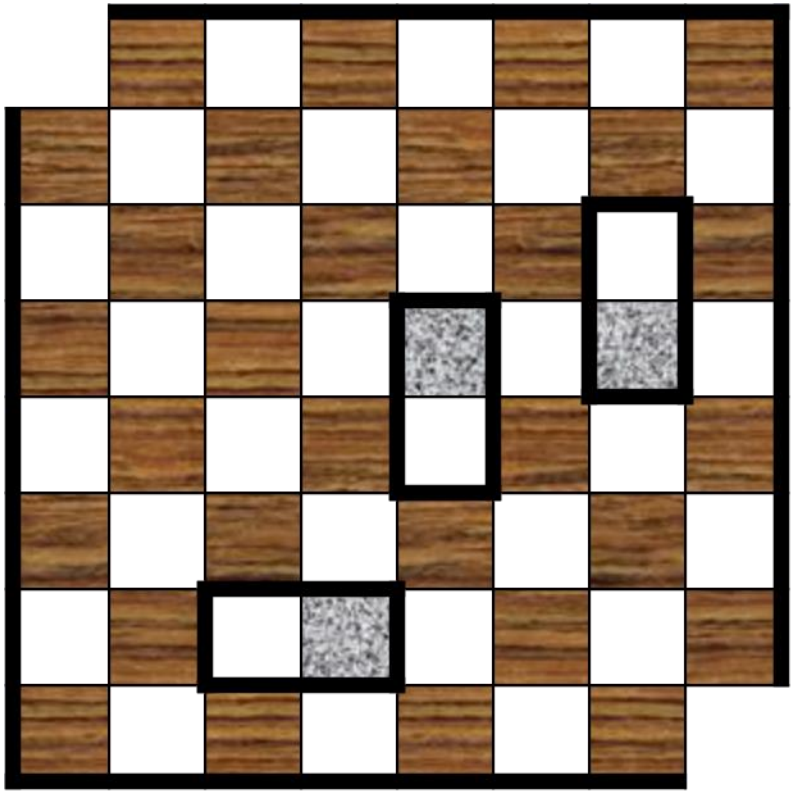
Pe fiecare linie sau coloană de pe tablă se va afla o singură regină. Se va parcurge tabla de şah linie cu linie (k=0..7), iar în cadrul unei linii coloană cu coloană (i=0..7) şi se vor plasa reginele în acele pătrate care nu sunt în "priza reginelor" plasate anterior. Pentru parcurgerea tablei se va utiliza tehnica backtracking.

Deoarece pe fiecare linie a tablei de şah se poate găsi exact o regină, o soluţie rezultat se poate reprezenta sub forma unui vector *C* = (*c*0,..., *c*7) unde *c[k]* reprezintă coloana pe care se află regina de pe linia *k* (c[k] aparține intervalului 0-7).

Metoda Backtracking – exerciții

- *Spaţiul soluţiilor posibile* este produsul cartezian $S = C \times C \times C \times C \times C \times C \times C \times C$
- *Condiţiile interne*, rezultă din regulile şahului şi sunt reprezentate de faptul că două dame nu se pot afla pe o aceeaşi coloană sau pe o aceeaşi diagonală.

Metoda Backtracking – exerciții



Metoda Backtracking – exerciții

Funcţia Solutie trebuie să verifice dacă nu există regine care se află pe aceeaşi coloană sau dacă nu cumva există regine care se atacă pe diagonală.

Verificarea este simplă. Trebuie să verificăm că intre elementele (c0, c1, c2, c3, c4, c5, c6, c7) nu există două care au aceeaşi valoare. Aceasta ar insemna că avem două regine pe aceeaşi coloană.

Apoi mai trebuie să verificăm că orice i,k din {0, 1, 2, 3, 4, 5, 6, 7}, |i-k| ≠ |ci-ck|. Aceasta este condiţia ca să nu existe două regine care se atacă pe diagonală. Verificări similare vor fi efectuate pe parcurs de funcţia Valid.

De data trecută: Backtraking

Eficienta algoritmilor

Timpul de executie

Generarea de numere aleatoare

Biblioteci software

Analiza algoritmilor

Algoritmii pot fi analizați din multe puncte de vedere:

- după criterii de performanță, cum sunt timpul de execuție sau memoria necesară, incluzând cazuri particulare, de exemplu cazul cel mai defavorabil, cazul cel mai des întâlnit în practică sau cazul cel mai favorabil
- după facilitățile suplimentare pe care algoritmul le furnizează. Exemple: indiferent de operațiile asupra ei, o colecție rămâne tot timpul sortată; pointerii la elementele unei colecții se pot folosi și după operații gen adăugare sau ștergere

Analiza algoritmilor

Algoritmii pot fi analizați din multe puncte de vedere:

- ținând cont de portabilitatea unei implementări: se utilizează doar facilități standard ale limbajului de programare sau sunt necesare instrucțiuni sau funcții care sunt disponibile doar pe un anumit compilator, platformă hardware sau sistem de operare
- după necesități administrative: unii algoritmi sunt mai simplu de implementat și testat, aspect important când există termene limită strânse care trebuie respectate

Big O notation

O notație des întâlnită pentru performanța unui algoritm este O(expr), unde *expr* este o funcție, în general exprimată în funcție de variabila **n**.
Notația *O(...)* (en: big O notation) reprezintă numărul de operații efectuate în cazul cel mai defavorabil, în funcție de numărul *n* de elemente de intrare.

Big O notation - exemple

Fie un vector cu dimensiune fixă, și *n* numărul curent de elemente din el. Câteva exemple despre cum se calculează *O(...)* pentru diverse operații cu acest vector:

- adăugarea unui element - **O(ct)**, deoarece indiferent de dimensiunea vectorului, este nevoie de un număr constant de operații (ex: *v[n++]=e;*)
- *ștergerea unui element* - **O(n)**, deoarece în cazul cel mai defavorabil (ștergerea primului element din vector), trebuie să mutăm toate celelalte *n* elemente la stânga cu o poziție. Strict vorbind, este nevoie de *n-1* operații, dar la limită, când *n* tinde la infinit, -1 devine nesemnificativ.

Big O notation - exemple

Fie un vector cu dimensiune fixă, și *n* numărul curent de elemente din el. Câteva exemple despre cum se calculează *O(...)* pentru diverse operații cu acest vector:

- inserarea unul element - O(n), deoarece în cazul cel mai defavorabil (inserarea pe prima poziție din vector), trebuie să mutăm toate celelalte *n* elemente la dreapta cu o poziție.
- căutarea unui element - O(n), deoarece în cazul cel mai defavorabil (elementul căutat nu este în vector), trebuie parcurse toate elementele vectorului

Big O notation - exemple

Exemple de algoritmi și cum crește numărul de operații necesare dacă n crește de la 10 la 100:

Tip algoritm	**Exemple de algoritmi**	**n=10**	**n=100**
Constant O(ct)	- adăugare de elemente în vector fix - calcul valoare absolută	1	1
Logaritmic $O(\log_2(n))$	- căutare binară într-un vector	3.3	6.6
Linear O(n)	- căutare sau ștergere într-un vector	10	100
Linearitmic $O(n*\log_2(n))$	- quicksort	33.2	664
Pătratic $O(n^2)$	- bubblesort	100	10000
Exponențial $O(2^n)$	- problema comis-voiajorului	1024	10^{30}

Big O notation - exemple

Pentru a ne da seama de diferențele dintre aceste tipuri de complexități, să presupunem că avem de sortat un vector de 1,000,000 elemente.

Folosind *quicksort*, am avea de efectuat ~20,000,000 de operații. Dacă un calculator execută 1,000,000 de asemenea operații/secundă, înseamnă că vom avea nevoie de 20 de secunde pentru sortare.

Dacă am fi folosit *bubblesort*, am avea de efectuat 1,000,000,000,000 operații, deci va fi nevoie de aproape 278 ore.

Big O notation - exemple

Pentru a ne da seama de diferențele dintre aceste tipuri de complexități, să presupunem că avem de sortat un vector de 1,000,000 elemente.

Folosind *quicksort*, am avea de efectuat ~20,000,000 de operații. Dacă un calculator execută 1,000,000 de asemenea operații/secundă, înseamnă că vom avea nevoie de 20 de secunde pentru sortare.

Dacă am fi folosit *bubblesort*, am avea de efectuat 1,000,000,000,000 operații, deci va fi nevoie de aproape 278 ore.

Omega (Ω) notation

Omega (Ω) notation este folosită pentru a descrie limita inferioară a timpului de execuție sau spațiului de memorie al unui algoritm.

De exemplu, dacă avem un algoritm cu complexitate Ω(n), acesta indică că timpul de execuție sau spațiul de memorie va crește cel puțin în mod liniar odată cu creșterea mărimii datelor de intrare.

Notația Omega este utilă pentru a oferi o limită inferioară a performanței unui algoritm și ne ajută să înțelegem că un algoritm nu poate fi mai rapid sau mai eficient decât o anumită valoare minimă.

Theta (Θ) notation

Theta (Θ) notation combină atât notația Big O (limita superioară) cât și notația Omega (limita inferioară).

Se folosește pentru a indica o limită strânsă asupra performanței algoritmilor, arătând că timpul de execuție sau spațiul de memorie este în mod aproximativ proporțional cu funcția specificată.

De exemplu, dacă un algoritm are complexitate Θ(n), atunci timpul de execuție sau spațiul de memorie vor crește în mod liniar odată cu creșterea mărimii datelor de intrare și există o corelație strânsă între acestea.

Notația Theta este folosită pentru a descrie complexitatea algoritmului într-un mod mai precis decât notația Big O singură, deoarece specifică atât limita superioară, cât și limita inferioară.

Analiza algoritmilor

```
for(i=0;i<n;i++)
        for(j=0;j<n;j++)
                a[i][j]=0;
```

Analiza algoritmilor

```
for(i=0;i<n;i++)
        if(v[n]==0)
                break;
```

Analiza algoritmilor - BubbleSort

```
void bubbleSort(int arr[], int n) {
    int i, j;
    int sorted = 0; // Variabilă pentru a verifica dacă mai sunt sortări de făcut
    for (i = 0; i < n-1; i++) {
        sorted = 1; // Presupunem că vectorul este sortat
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                sorted = 0; // Încă mai sunt sortări de făcut
            }
        }
        if (sorted) {
            break; // Dacă vectorul este sortat, ieșim din buclă
        }
    }
}
```

Analiza algoritmilor - BubbleSort

Complexitatea O (Big O):

- În cel mai rău caz (worst case), complexitatea timpului de execuție al algoritmului Bubble Sort este O(n^2). Aceasta înseamnă că numărul de operații crește cu pătratul mărimii vectorului.

Complexitatea Omega (Ω):

- În cel mai bun caz (best case), atunci când vectorul este deja sortat, complexitatea timpului de execuție al algoritmului Bubble Sort este Omega(n). În această situație, algoritmul va parcurge vectorul o singură dată pentru a verifica că este sortat și nu va face nicio interschimbare, deoarece nu este necesar.

De data trecută: Backtraking

Eficienta algoritmilor

Timpul de executie

Generarea de numere aleatoare

Biblioteci software

Masurarea timpului de executie

Pentru o analiza exactă a timpului de execuție al uniu program sau a unei porțiuni din program se poate folosi functia clock(), declarata in time.h.

In mod uzual se apeleaza clock() la inceputul si sfarsitul portiunii de analizat, se scad valorile si converteste in timp-real, prin impartire la CLOCKS_PER_SEC (numarul de "clocks" ai procesorului).

Masurarea timpului de executie

```
#include <time.h>

clock_t start, end;
double cpu_time_used;

start = clock();
... /* Do the work. */
end = clock();

cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

https://www.geeksforgeeks.org/rand-and-srand-in-ccpp/

Masurarea timpului de executie

Este important să rețineți că măsurarea timpului de execuție al unui algoritm poate fi influențată de mai mulți factori, cum ar fi specificațiile hardware ale sistemului și sarcinile în execuție pe computer.

Este recomandat să efectuați mai multe măsurători și să obțineți o medie a rezultatelor pentru a obține o valoare mai reprezentativă a timpului de execuție al algoritmului.

Masurarea timpului de executie

Aspecte ce tin de hardware (mecanisme de memorie cache, lucrul cu discul de memorie) si/sau sistem de operare (multithreading) pot influenta semnificativ rezultatele.

Se recomanda reluarea analizei in anii superiori dupa parcurgerea disciplinelor cu accent pe arhitectura sistemelor de calcul si/sau sisteme de operare.

De data trecută: Backtraking

Eficienta algoritmilor

Timpul de executie

Generarea de numere aleatoare

Biblioteci software

Generarea de numere aleatoare

Pentru a testa un algoritm e util să putem genera date de intrare potrivite în mod rapid (citirea de la tastatură presupune timp la fiecare rulare, citirea din fișier presupune crearea și popularea cu date a fișierului înainte de execuția programului).

Pentru a genera numere aleatoare în limbajul C, puteți utiliza funcțiile srand() și rand() din biblioteca <stdlib.h>.

Generarea de numere aleatoare

Pentru a testa un algoritm e util să putem genera date de intrare potrivite în mod rapid (citirea de la tastatură presupune timp la fiecare rulare, citirea din fișier presupune crearea și popularea cu date a fișierului înainte de execuția programului).

În general, utilizarea numerelor aleatoare adaugă elementul de imprevizibilitate, diversitate și explorare în aplicațiile și algoritmii noștri, contribuind la crearea de rezultate mai realiste, securizate și eficiente.

Generarea de numere aleatoare

Pentru a genera numere aleatoare în limbajul C, puteți utiliza funcțiile srand() și rand() din biblioteca <stdlib.h>.

Functia rand() returneaza la fiecare apel un numar natural cuprins intre [0, RAND_MAX).

În majoritatea implementărilor, valoarea RAND_MAX este setată la cel puțin 32767. Cu toate acestea, poate fi și o valoare mai mare, în funcție de implementarea specifică. De aceea, este recomandat să verificați documentația sau să consultați specificațiile limbajului C pentru a afla valoarea exactă a RAND_MAX în implementarea pe care o utilizați.

Generarea de numere aleatoare

Initializarea generatorului de numere pseudoaleatoare se poate face folosind functia srand(unsigned) care primeste un *seed* si permite generarea de secvente pseudoaleatoare distincte, intre rulari succesive. Apelul la srand() trebuie facut o singura data, ca parte a rutinei de initializare, inainte de orice apel la rand().

O practica uzuala este utilizarea rezultatului functiei time(0), care returneaza o data de tipul time_t, cu valoare distincta la fiecare apel (timpul curge unidirectional) si garanteaza ca la fiecare apel se obtine o alta secventa pseudoaleatoare. Astfel se va folosi srand(time(0));

Generarea de numere aleatoare

```
// genereaza o secventa de numrere pseudoaleatoare,
distincta la fiecare apel
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
   // Foloseste ora curenta pentru initializarea PRNG-ului
   srand(time(0));
   for(int i = 0; i<4; i++){
      printf(" %d ", rand());
      }
   return 0;
}
```

De data trecută: Backtraking

Eficienta algoritmilor

Timpul de executie

Generarea de numere aleatoare

Biblioteci software

Biblioteci software

Crearea si gestionarea de biblioteci software in C

Se foloseste pentru a "impacheta" mai multe fisiere obiect intr-un fisier care spre exemplu permite reutilizarea de functionalitate (spre exemplu fiserul biblioteca pentru operatii matematice are antetul declarat in fisierul math.h).

Biblioteci software

Crearea si gestionarea de biblioteci software in C

Se foloseste pentru a "impacheta" mai multe fisiere obiect intr-un fisier care spre exemplu permite reutilizarea de functionalitate (spre exemplu fiserul biblioteca pentru operatii matematice are antetul declarat in fisierul math.h).

Biblioteci software

Pas 1 : cream fisierul hs_utils.c cu urmatorul continut:

```
/* hs_utils.c */
unsigned estePar(unsigned long long n) {
    if (n%2==0){
      return 0;
    }
    //restul codului
}
```

Biblioteci software

Pas 2 : cream fisierul hs_utils.h in care declaram functiile din hs_utils.c:

```
/* hs_utils.h */
unsigned estePar(unsigned long long);
```

Biblioteci software

Pas 3: Putem folosi biblioteca creata mai sus intr-un program in felul urmator:

```
/* main.c */
#include "hs_utils.h"
void main(void) {
   if (estePar (23)){
     //...
   }
}
```

in care solicitam includerea la preprocesare a fisierului hs_utils.h care permite compilatorului sa aiba acces la antetul functiei estePrim urmand ca implementarea ei sa fie accesibila la linkare din fisierul obiect obtinut prin compilarea lui hs_utils.c.

Biblioteci software

La compilarea programului în mod direct va trebui să specificăm toate fișierele .c astfel:

gcc –Wall –o executabil main.c hs_utils.c

Pentru automatizarea regulilor de build se poate folosi utilitarul make.

Vă mulțumesc!