Proiectarea Algoritmilor 2009-2010

Laborator 0

Introducere și relaxare

Cuprins

1. De ce PA ?

Răspunsul este simplu: complexitatea algoritmilor, plăcerea de a explora alternative, de a le compara și în final de a rezolva optim problemele, posibilitatea de a fi creativ și de ce nu chiar inovativ, capacitatea de a generaliza anumite aspecte și de soluționa probleme din viața de zi cu zi.

Nu ne rezumăm la calcul computațional, algoritmii pot fi aplicați în orice circumstanțe și fundamentează o serie de decizii care guvernează multiple aspecte din realitatea înconjurătoare. Absolut toți algoritmii analizați pot fi regăsiți în viața de zi cu zi, într-adevăr cu frecvențe diferite, iar faptul că noi putem înțelege ce se întâmplă în profunzime, consider că ne poate da acel gram de unicitate care ne diferențiază.

Astfel, avem o plajă largă din care să alegem: în cazul *rețelisticii*: algoritmii de rutare, politicile de load-balancing si flux maxim; în *sisteme de operare*: scheduling, caching; *grafica* cu prelucrarea imaginilor, determinarea contururilor, iluminare și texturare; *baze de date* cu indecși sub formă de Arbori B, operatorul ”select” în sine; *componente hardware* pornind de la banalul incrementator pe biți și până la procesoarele DSP pentru prelucrarea semnalelor; *inteligența artificială* cu tot ce înseamnă prelucrarea limbajului natural, analiza rețelelor sociale, analiză semantică, sisteme bazate pe reguli. Și lista poate fi continuată lejer, ajungând la dimensiuni considerabile. Practic, jocul se rezumă

la următoarea regulă: spune orice domeniu cu o problemă specificată şi sigur găsim un algoritm care să ajute, măcar la nivel de euristică, dacă o soluţie optimă nu este fezabilă.

Şi să fim sinceri: dacă acum nu incercam sa le intelegem si sa le stapanim, atunci când?

Astfel accentul este pus pe partea formală, importanţa şi utilităţile algoritmilor, iar implementările propriu-zise pot fi considerate o „validare”.

Ultimul aspect cu adevărat important este că algoritmii stau la baza a cam tot ce înseamnă Computer Science, iar proiectarea eficientă a acestora face diferenţa.

2. Pentru cine ?

În primul rând pentru tine şi sperăm să conştientizezi acest lucru: totul se rezumă la a-ţi construi o bază de cunoştinţe pe care să le foloseşti în viaţa, indiferent de ce specializare vei urma. Oricum nu există domeniu din computer science în care să nu foloseşti un algoritm, oricât de simplu sau complex. Pe de altă parte, numărul de persoane care pot spune că înţeleg algoritmii, ştiu să-i aplice şi să determine soluţia optimă este restrâns, creând astfel o comunitate activă si dinamică.

3. Ce vom face ?

Vom aprofunda elementele fundamentale necesare rezolvarii oricărei probleme, prezentand mai multi algoritmi de rezolvare pentru fiecare problemă studiată şi evidenţiând algoritmii optimi. Pornind de la aceste elemente, vom accentua punctele de interes identificate, descoperind şabloane de rezolvare şi modalităţi de constuire a soluţiilor pentru o problemă.

Câteva din aspectele atinse includ:

- Divide et Impera, Greedy şi programare dinamică;
- Parcurgeri pe grafuri (BFS, DFS) şi sortare topologică;
- Căutări in spaţiul stărilor (best-first, A*);
- Alte aplicaţii DFS (componente tare conexe, puncte de articulaţie, punţi);
- Backtracking şi optimizări (prospective şi retroactive);
- Drumuri minime şi arbori minimi de acoperire;
- Algoritmi Minimax;
- Fluxuri maxime;
- Algoritmi aleatori.

4. Ce nu vom face ?

Copiat teme, laboratoare. Pentru a continua ideea simplă şi pentru a accentua întrebarea pe care trebuie să ne-o punem cât mai des – de ce? -, formulăm un răspuns la fel de simplu şi imediat: fiindcă este greşit. Analogia cea mai simplă pe care o putem face este cu viaţa per ansamblu la nivelul căreia întotdeauna există un echilibru: ce faci întotdeauna ţi se întoarce, cu diverse ponderi, în unele cazuri putând fi chiar înzecit. Şi dacă la o materie care îţi stimulează creativitatea, inovaţia, te pune la

încercare și care fundamentează abordările din Computer Science apelezi la astfel de metode, sincer trebuie să îți pui întrebarea: *eu ce caut la această facultate*?

Din punctul nostru de vedere nu copiezi din mândrie personală, la urma urmei este bine să fii deasupra celora care apelează la astfel de metode. În plus să nu uităm satisfacția personală ulterioară. Și ce dacă nu sunt prinși? Garantăm că tot ei au numai de pierdut, atât din prisma aspectelor învățate, cât și a implicațiilor ulterioare.

Alegerea acestei facultăți a fost rezultatul unei documentări minuțioase care a dus la identificarea acesteia cu idealurile personale ale fiecăruia dintre voi. Rolul pe care şi-l asumă un îndrumător este doar de a materializa acest vis si de a va ajuta in atingerea telurilor propuse la intrarea in facultate. Metodele de copiat reprezintă mijloace care alterează atingerea obiectivelor personale si a perspectivei despre nivelul si capacitățile dvs. Totodată acestea nu pot sa influențeze perspectiva posibililor angajatori, in momentul trecerii in practica a cunoştinţelor acumulate in perioada facultății. Astfel, singurul rezultat este pe termen scurt si reflecta zicala „ti-ai furat singur căciula”.

În plus, în concordanță cu principiul „carrot and stick” va exista o verificare minuțioasă împotriva copierii. Pe scurt, *la prima abatere – -10 (minus 10) pentru tema respectivă, la a doua restanță*. Regulile sunt foarte simple.

5. Evaluarea temelor

Notarea temelor va avea la bază următoarele criterii și punctaje aferente:

- 5 puncte dacă se respectă cerințele temei, adică compilare și execuție fără erori astfel încât să se obțină rezultatele cerute;
- 3 puncte pentru o implementare eficientă;
- 1 punct pentru comentariile din fișierele sursă, respectiv ReadMe;
- 1 punct pentru claritatea codului, lizibilitate.

Pentru a asigura o evaluare uniformă a temelor, va exista o singură persoană care este responsabilă de corectarea unei teme la nivelul întregii serii.

Toate temele vor fi corectate folosind același set de teste. Pentru a veni in sprijinul vostru și pentru a testa corectitudinea și eficiența implementării temelor, fiecare enunț va avea un link către o pagină de pe *infoarena* ([12]) unde se pot testa rezolvările realizate în C/C++. Temele rezolvate în Java (sau alte limbaje de programare, dar cu *acordul explicit* al responsabilului de temă) nu vor putea fi testate folosind infoarena. Pentru aceste teme, aveți posibilitatea să folosiți direct *vmchecker* [13], putând astfel să beneficiați de feedback înainte de trimiterea rezolvării temei.

Toate temele vor fi testate automat folosind *vmchecker*, dar și prin verificarea codului și citirea Readme-ului.

Mai multe detalii in regulament PA 2010.

6. Coding style

Stilul de „redactare” al programelor are impact major asupra celor care parcurg respectivul cod. Pentru compilator nu este nici o diferență, dar totuși să nu fim egoiști când scriem. În plus, de cele mai multe ori ne ajută inclusiv pe noi, oferind claritate, conciziune, viteză mult mai mare de regăsire și reamintire a anumitor aspecte, toate fiind posibile prin anumite elemente identificate la nivelul unui stil bun de codare.

Totodată, stilul de codare ține și de gradul de profesionalism care se dorește a fi exprimat, marcând astfel coehesiunea ideilor, structurarea acestora, claritatea și ordonarea din perspectiva organizării personale. Astfel, un cod scris dezordonat și greu lizibil duce inevitabil la o catalogare a persoanei responsabile drept delăsătoare, dezordonată și chiar incoerentă datorită neregăsirii unor elemente; și cu toți știm cât de importantă este prima impresie.

Desigur există particularități specifice fiecărui limbaj de programare, precum și individuale din prisma preferințelor personale. Ulterior pot apărea constrângeri generate de specificul echipei sau companiei din care veți face parte, dar există un set general de reguli care merită urmat (exemplele au fost preluate de pe Wikipedia):

Prezentare generală

- Formatare (indentare) generală

```
if (hours < 24 && minutes < 60 && seconds < 60) {
   return 1;
} else {
   return 0;
}
```

- Spații în cadrul structurilor și între operatori

```
int i;
for (i = 0; i < 10; ++i) {
      printf("%d", i * i + i);
}
```

- Tab-uri pentru aliniere

```
int         ix;         // Index
double      sum;        // Accumulator pentru suma
```

Nume variabile/funcții, logică precum și alte tehnici

- Nume adecvate și sugestive pentru variabile/funcții

```
int correctFormat (int hours, int minutes, int seconds) {
   if (hours < 24 && minutes < 60 && seconds < 60)
      return 1;
   else
      return 0;
}
```

în loc de:

```
void correctFormat (int a, int b, int c) {
   if (a < 24 && b < 60 && c < 60)
      return 1;
   else
```

```
      return 0;
}
```

- Valorii boolene în structuri de decizie, doar pentru efect stilistic întrucât codul generat alternativ este oricum optimizat de compilator:

```
return (hours < 24) && (minutes < 60) && (seconds < 60);
```

- Comparații de tip Left-hand:

```
if ( a = 42 ) { ... }   // Eroare inadvertentă care poate apărea și e
                        dificil de identificat
if ( 42 = a ) { ... }   // Eroare de compilare
```

- Cicluri și structuri de control – utilizarea acoladelor și indentarea corespunzătoare nivelului de imbricare (evitarea problemelor care pot apărea de exemplu în Python datorate indetării greșite):

```
for (int i = 0; i < 5; i++) {
   printf("%d", i * 2);
}

print "Ended loop";
```

- Liste – elementele sunt plasate pe linii diferite:

```
const char *zile[] = {
   "luni",
   "marti",
   "miercuri",
   "joi",
   "vineri,
};
```

Caracteristici specifice fiecărui limbaj se regăsesc în referințele [4..10].

Alte aspecte care trebuie luate în vedere la scrierea elegantă a codului:

- **modularitatea programelor** (dimensiune recomandată este de 20 de linii per funcție/procedură), totodată asigurând creșterea lizibilității codului, a posibilităților de reutilizare; se dorește de asemenea obținerea unei cuplări strânse în cadrul aceluiași modul, precum o cuplare cât mai slabă între module diferite din perspectiva proiectării arhitecturale;
- **folosirea de constante/variabile** în loc de valori hard-coded;
- **reutilizarea codului**, evitând astfel situații de repetare de tip „copy paste” a unor secțiuni de cod, care în urma unor modificări ulterioare pot duce la anumite situații dificile de tip debugging.

7. Accentul pe algoritmi

După cum reiese și din denumirea materiei, accentul va fi pus pe partea formală și înțelegerea efectivă a algoritmilor.

Astfel, la nivelul implementării, limbajul ales este la latitudinea fiecăruia. Din perspectiva uşurinţei de scriere, a structurilor de date disponibile şi a sintaxei se recomandă Java.

Din perspectiva performanţelor obţinute, recomandările înclină spre C/C++, cu amendamentul că pentru structuri de bază să fie integrat **STL** (*Standard Template Library* - http://www.sgi.com/tech/stl/download.html). Acesta pune la dispoziţie containere pentru date (Secvenţiale – Vectori, Liste, Cozi Dublu Înlănţuite sau asociative – Maps, Sets), structuri particulare (Cozi, Stive, Cozi de prioritate, Bitsets, Valarrays), algoritmi (căutare binară, sortare), iteratori şi managementul memoriei ([1], [2] şi [3]).

Astfel, prezentăm un exemplu pentru utilizarea vectorilor:

```
vector<string> SS;

SS.push_back("The number is 10");
SS.push_back("The number is 20");
SS.push_back("The number is 30");

cout << "Loop by index:" << endl;

int ii;
for(ii=0; ii < SS.size(); ii++)
{
   cout << SS[ii] << endl;
}

cout << endl << "Constant Iterator:" << endl;

vector<string>::const_iterator cii;
for(cii=SS.begin(); cii!=SS.end(); cii++)
{
   cout << *cii << endl;
}

cout << endl << "Reverse Iterator:" << endl;

vector<string>::reverse_iterator rii;
for(rii=SS.rbegin(); rii!=SS.rend(); ++rii)
{
   cout << *rii << endl;
}

cout << endl << "Sample Output:" << endl;

cout << SS.size() << endl;
cout << SS[2] << endl;

swap(SS[0], SS[2]);
cout << SS[2] << endl;
```

cu output-ul aferent:

```
Loop by index:
The number is 10
The number is 20
The number is 30

Constant Iterator:
```

```
   The number is 10
   The number is 20
   The number is 30

   Reverse Iterator:
   The number is 30
   The number is 20
   The number is 10

   Sample Output:
   3
   The number is 30
   The number is 10
```

Pentru utilizarea unei liste simplu înlănțuite se poate folosi următorul exemplu:

```
   list<int> L;
   L.push_back(0);              // Insert a new element at the end
   L.push_front(0);             // Insert a new element at the beginning
   L.insert(++L.begin(),2);     // Insert "2" before position of first
argument
                                // (Place before second argument)
   L.push_back(5);
   L.push_back(6);

   list<int>::iterator i;

   for(i=L.begin(); i != L.end(); ++i)
      cout << *i << " ";
   cout << endl;
   return 0;
```

în urma căruia se obține:

```
   0 2 0 5 6
```

Un aspect esențial cu care este bine să vă familiarizați (dacă nu ați făcut-o deja) este **debugging-ul**. Premisa de la care plecăm este diferența majoră dintre un program care funcționează și unul incomplet sau incorect datorită simplului fapt că nu a fost supus unei proceduri de testare suficient de minuțioase și că nu a fost „debugged” suficient.

Acum, pentru a realiza debugging-ul avem la dispoziție:

- tool-uri cu funcționalități de debugging pentru C (Windows și Linux) și Java: Eclipse, Netbeans sau Visual Studio au deja integrate un debugger;
- printf debugging cu simpla afișare a rezultatelor parțiale și determinarea punctului în care funcționarea devine necorespunzătoare;
- log debugging care presupune înregistrarea unor evenimente (într-un fișier, la consolă) și ulterior consultarea acestora pentru determinarea punctului în care s-a produs eroarea (de ex. Log4j pentru Java).

În altă ordine de idei, folosirea unui **IDE** pentru dezvoltare este mai mult decât recomandată: Eclipse cu plugin-uri aferente fiecărui limbaj, NetBeans pentru Java, Visual Studio pentru C, C++, C# pentru a spori eficiența scrierii efective de cod (intellisense, auto-completion, code generation, code formatting and refactoring).

8. Aplicaţii de laborator (socializare şi prezentare generală PA)

Responsabil laborator: Mihai Dascălu (mihai.dascalu@cti.pub.ro)

9. Referinte

[1] Standard Template Library Programmer's Guide

http://www.sgi.com/tech/stl/

[2] C++ Standard Template Library

http://www.cppreference.com/wiki/stl/start

[3] C++ STL (Standard Template Library) Tutorial and Examples

http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html

[4] Code Conventions for the Java Programming Language

http://java.sun.com/docs/codeconv/

[5] Google C++ Style Guide

http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

[6] C Coding Standard

http://micrium.com/newmicrium/uploads/file/appnotes/an2000.pdf

[7] C++ Layout and Comments

http://geosoft.no/development/cppstyle.html#Layout

[8] Brad Abrams, Design Guidelines, Managed code and the .NET Framework

http://blogs.msdn.com/brada/articles/361363.aspx

[9] Mozilla Coding Style Guide

https://developer.mozilla.org/En/Mozilla_Coding_Style_Guide

[10] PHP::PEAR Coding Standards

http://pear.php.net/manual/en/standards.php

[11] Gustav Käser Training International

http://www.gustavkaeser.com/www/pages/ro/

[12] Infoarena

http://infoarena.ro/teme

[13] VMChecker

http://svn.rosedu.org/vmchecker, http://github.com/vmchecker/vmchecker