Laborator 12 : Algoritmi aleatori

Obiective laborator

- Înțelegerea noțiunilor de bază legate de algoritmii aleatori – Algoritmi Las Vegas și Monte Carlo;
- Familiarizarea cu rezolvarea folosind algoritmi aleatori a problemelor clasice;
- Diversificarea perspectivei de analiză și rezolvare a problemelor.

Aplicații practice

Algoritmii aleatori se împart în principal în 2 clase:

- Algoritmi care rezolvă probleme de optim: soluția calculată de algoritm este garantat corectă, dar este aproximativă (nu este optimală). În acest caz, soluția suboptimală este considerată acceptabilă având o marjă de aproximare controlată probabilistic – **algoritmi de aproximare, algoritmi genetici și algoritmi aleatori de tip Las Vegas**;
- Algoritmi care rezolvă o problema ce acceptă o singură soluție: se renunță la exactitatea rezolvării preferându-se o soluție rapidă care se apropie cu o probabilitatea suficient de mare de soluția exactă – corectitudinea **nu** este garantată – **algoritmi aleatori de tip Monte Carlo și stocastici (Markov)**.

Printre implicațiile practice ale algoritmilor aleatori se numără:

- optimizarea diverșilor algoritmi, în general în vederea asigurării dispersiei corespunzătoare a valorilor;
- diverse inițializări (ex. Algoritmi Genetici pentru indivizi) sau selecții de date după o distribuție prestabilită (în general Gaussiană);
- reducerea complexității unor probleme specifice.

Descrierea problemei și a rezolvărilor

Primele aspecte care trebuie clarificate sunt caracteristicile algoritmilor aleatori:

- necesitatea micșorării timpului de rezolvare a problemei prin relaxarea restricțiile impuse soluțiilor;
- este suficientă o singură soluție care se apropie cu o probabilitate măsurabilă de soluția exactă;
- în final se poate obține o soluție suboptimală cu o marjă de eroare garantată prin calcul probabilistic.

Generatorul de numere aleatorii se află la baza construcției și funcționării algoritmilor aleatori. Astfel, pentru rulări diferite există șansa ca algoritmul să se comporte diferite, chiar dacă datele de intrare, respectiv rezultatele sunt aceleași. Astfel, pentru același set de date de intrare, algoritmii familiei se comportă diferit, chiar dacă rezultatele sunt aceleași.

Algoritmi Las Vegas

Caracteristici:

- Determină soluția corectă a problemei, însă timpul de rezolvare nu poate fi determinat cu exactitate;
- Creșterea timpului de rezolvare implică creșterea probabilității de terminare a algoritmului;
- După un timp infinit se ajunge la soluția optimă și algoritmul se termină sigur;
- Probabilitatea de găsire a soluției crește extrem de repede încât să se determine soluția corectă într-un timp suficient de scurt.

Complexitate teoretică:

```
f(n)=O(g(n)) daca ∃c>0, n0 > 0 a.i.:
```

- $\forall n>n0$ $0 < f(n) < c\alpha g(n)$ cu o probabilitate de cel puțin $1 - n^{-\alpha}$, α fixat și suficient de mare.

Implicații:

Procentul algoritmilor Las Vegas care consumă cel mult $c\alpha g(n)$ resurse de calcul din totalul unei familii de algoritmi de complexitate O(g(n)) este $1 - n^{-\alpha}$. Pentru α suficient de mare există șanse foarte mici să se folosească un algoritm al familiei care nu respectă limita de complexitate.

Problemă:

- Capitolele unei cărți sunt stocate într-un fișier text sub forma unei secvențe nevide de linii;
- Fiecare secvență este precedată de o linie contor ce indică numărul de linii din secvență, iar specificul indică fiecare astfel de secvență este lungă;
- Fiecare linie din fișier este terminată prin CR,LF;
- Toate liniile din secvență au aceeași lungime;
- Fiecare secvență de linii conține o linie (titlul capitolului) ce se repetă și care apare în cel puțin q = 10% din numărul de linii al secvenței.

Cerință:

- Pentru fiecare secvență de linii să se tipărească titlul capitolului (linia care se repetă).

Complexitate variantă iterativă: $O(n^2)$ în cazul cel mai defavorabil

Rezolvare aleatoare:

```
selectie_linii(n,Secv) // Pp n = dim secv > 100
    while (1)
        i = random(0,n-1) // selectez o linie
        j = random(0,n-1) // si inca una
        if (i != j && linie(i,Secv) = linie(j,Secv)) // le compar
            return linie(i,Secv) // am gasit linia
```

Complexitate variantă aleatoare: $O(lg^{-1}(1/a)lg(n))=O(lg(n))$, unde a = 1 - q(q-1)/10000, q=10 – probabilitatea de regăsire a titlului capitolului.

Observații:

De exemplu pentru n=100 și q=10%, după 3500 de iterații, probabilitatea ca soluția să fie corectă poate fi considerată 1; dacă q=30%, atunci numărul de iterații devine 500. Aproprierea probabilității de 1 este atât de mare încât precizia de calcul cu 12 zecimale nu mai asigură obținerea valorii exacte și, practic, terminarea algoritmului devine certă.

Algoritmul se comportă foarte bine chiar și atunci când în condițiile teoretice nu sunt respectate întrucât avem de-a face cu numere pseudo-aleatorii și secvența de linii nu este formată aleator.

Algoritmi Monte Carlo

Caracteristici:

- Determină o soluție a problemei care e garantat corectă doar după un timp infinit de rezolvare – soluție aproximativă;
- Presupun un număr finit de iterații după care răspunsul nu este garantat corect;
- Creșterea timpului de rezolvare implică creșterea probabilității ca soluția găsită să fie corectă;
- Soluția găsită într-un timp acceptabil este aproape sigur corectă (există o probabilitate mică ca soluţia să nu fie corectă).

Complexitate teoretică:

```
f(n)=O(g(n)) daca ∃c>0, n0 > 0 a.i.:
```

$\forall n>n0\ 0 < f(n) < c\alpha g(n)$ cu o probabilitate de cel puțin $1 - n^{-\alpha}$, α fixat și suficient de mare.

- Probabilitatea ca soluția determinată de algoritm să fie corectă este de cel puțin $1 - n^{-\alpha}$.

Implicații:

Procentul algoritmilor Monte Carlo care consumă cel mult $c\alpha g(n)$ resurse de calcul din totalul unei familii de algoritmi de complexitate O(g(n)) pentru a găsi o soluție corectă cu o probabilitate de cel puțin $1 - n^{-\alpha}$ este $1 - n^{-\alpha}$. Pentru α suficient de mare există șanse foarte mici să se folosească un algoritm al familiei care nu respectă limita de complexitate și nu se termină cu o soluție corectă.

Problemă:

- Testarea dacă un număr dat n este prim.

Complexitate variantă clasică: $O(\sqrt{n})=O(^{k/2})$ unde k = nr. de biți ocupați de n

Rezolvare aleatoare folosind teorema lui Fermat (Dacă n este prim atunci pentu $\forall\ 0 < x < n$, $x^{n-1} \bmod n = 1$):

```
prim1(n,α) // detectează daca n e număr prim
    if (n <= 1 || n mod 2 == 0) return false
    limit = limita_calcul(n,α) //nr min paşi pt sol corectă cu P=1-n^-α
    for (i = 0 ; i < limit ; i++)
        x = random(1,n-1) // aleg un număr oarecare
        if (pow_mod(x,n) != 1) return false // T. Fermat
    return true

pow_mod(x,n) // calculează xn-1 mod n
    r = 1
    for (m = n - 1 ; m > 0 ; m = m / 2)
        if (m mod 2 != 0) // testez daca m e par sau nu
            r = x*r mod n
```

```
        x = (x*x) mod n
    return r;
```

Problema acestei abordări constă în faptul că nu putem stabili cu exactitate care este limita de calcul.

Pornind de la următoarea teoremă: Pentru orice număr prim ecuația x^2 mod n = 1 are exact 2 soluții: x_1 egal 1 și x_2 egal n – 1, obținem următoarea definiție pentru X = martor al divizibilității lui n : Fie n > 1 și 0 < x < n două numere astfel încât x^{n-1} mod n != 1 sau x^2 mod n != 1, x != 1 si x != n – 1.

```
prim2(n,α)
    if(n <= 1 || n mod 2 == 0) return false
    limit = limita_calcul(n,α)
    for (i = 0 ; i < limit ; i++)
        x = random(1, n-1)
    if (martor_div(x,n)) return false
    return true;

martor_div(x,n) // determina daca X=martor al divizibilitatii lui n
    r = 1; y = x;
    for(m = n - 1 ; m > 0 ; m = m / 2) // puterea
        if (m mod 2 != 0) // putere impara
            r = y * r mod n
        z = y // salvez valoarea lui y
        y = y * y mod n // calculez y2 mod n
        if (y == 1 && z != 1 && z != n-1)//verific teorema anterioară
            return 1
    return r != 1 // teorema Fermat
```

Complexitate: $O(lg^2n)=O(k^2)$

Concluzii și observații

Metodele descrise pot fi aplicate și se adresează unei plaje largi de probleme, iar abordările prezentate pot duce la scăderi drastice a timpilor de execuție.

Referințe

[1] C. Giumale – Introducere în Analiza Algoritmilor – cap. 6.1

[2] T. H. Cormen & all – Introducere în algoritmi – cap. 8.3, 1990

[3] http://www.soe.ucsc.edu/classes/cmps102/Spring04/TantaloAsymp.pdf [http://www.soe.ucsc.edu/classes/cmps102/Spring04/TantaloAsymp.pdf]

[4] http://www.mersenne.org/ [http://www.mersenne.org/]

Probleme

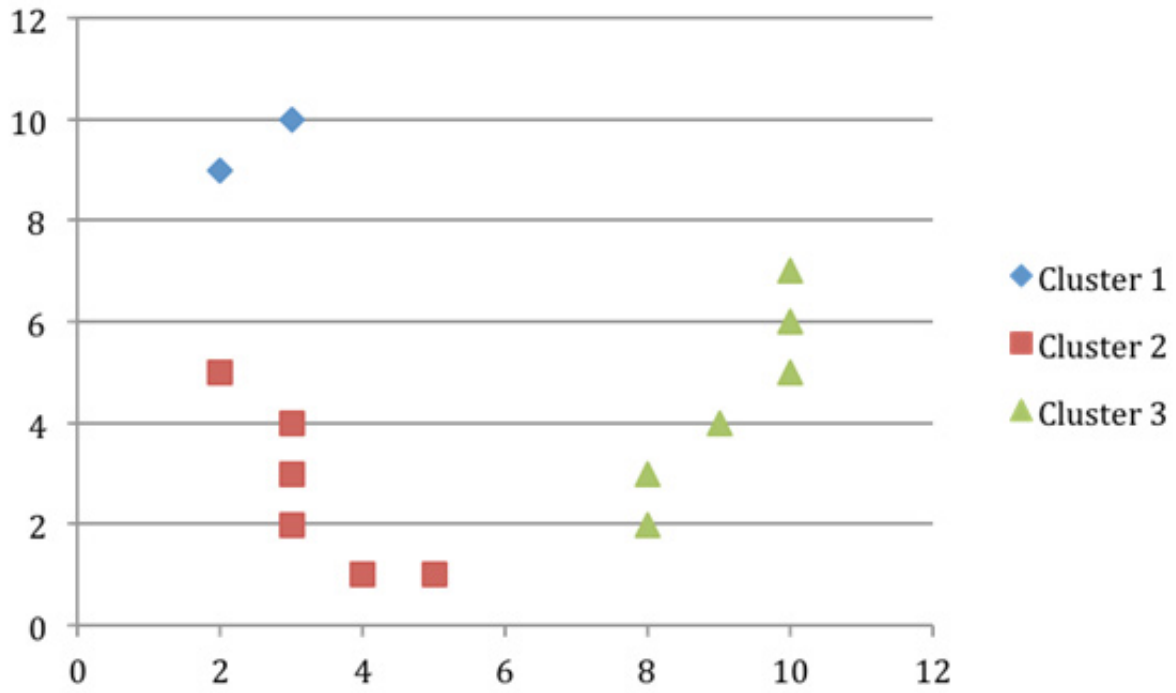
K-means

Fie n puncte într-un spațiu bidimensional. Se dorește o grupare a acestora în k clustere - un grup de puncte situate într-o vecinătate spațială care să maximizeze coeziunea intra-cluster și să asigure o cuplare slabă inter-clustere.

Spre exemplu, pentru setul de puncte:

```
(2, 5), (2, 9), (3, 2), (3, 3), (3, 4), (3, 10), (4, 1), (5, 1),
(8, 2), (8, 3), (9, 4), (10, 5), (10, 6), (10, 7)
```

Se poate observa o grupare naturală în 3 clustere:



Coeziunea internă este determinată drept 1 / media distanțelor către clustroid (punctul din cluster cel mai apropiat de toate celelalte noduri), iar cuplarea inter-clustere este 1 / distanța dintre clustere (distanța dintre clustroizii aferenți). Pașii algoritmului k-Means sunt următorii (pentru N și k date):

1. Se selectează k puncte random din spațiu care vor fi centroizii inițiali ai clusterelor.
2. Se efectuează iterativ următorii pași cât timp atribuirile fiecărui nod la un cluster rămân neschimbate:
    a. Asignarea fiecărui nod unui cluster (distanța minimă euclidiană către centroizii din pasul curent este minimă);
    b. Recalcularea centroidului drept media aritmetică a coordonatelor punctelor asignate.

Pentru un set de date și un k stabilit se vor determina clusterele aferente. Se va implementa și o optimizare a selecției inițiale de puncte pornind de la principiul că acestea ar trebui să fie geografic cât mai dispersate (se dorește maximizarea distanței între centroizii inițiali; mai multe detalii la [1]).

[1] http://en.wikipedia.org/wiki/K-means%2B%2B#Initialization_algorithm [http://en.wikipedia.org/wiki/K-means%2B%2B#Initialization_algorithm]