Proiectarea Algoritmilor

Curs 12 – Euristici de explorare (continuare)
Algoritmi aleatorii

Bibliografie

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

[2] Cormen – Introducere in algoritmi – cap. 8.3

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

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

Proiectarea Algoritmilor

Euristici de explorare (continuare)

Reminder Algoritmul A* - completitudine și optimalitate

- Definiție 7.2: Funcția euristică h este admisibilă dacă pentru orice nod n din spațiul stărilor $h(n) \leq h^*(n)$. Cu alte cuvinte, o euristică admisibilă h este optimistă și $h(\gamma) = 0$ pentru orice nod $\gamma \in \Gamma$.

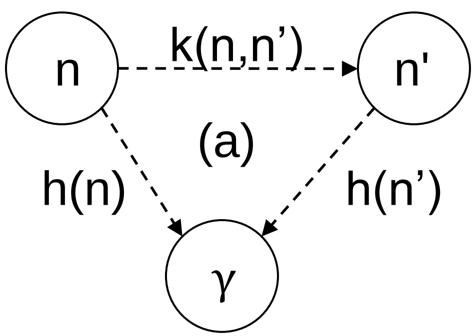
- Teorema 7.2: Algoritmul A* ghidat printr-o euristică admisibilă descoperă soluția optimă dacă există soluții.

PA

Euristici – consistență și monotonie

- Definiție 7.4: O euristică h este consistentă dacă pentru oricare două noduri n și n’ ale grafului explorat, astfel încât n’ este accesibil din n, există inegalitatea: $h(n) \leq h(n') + k(n,n')$, unde $k(n,n')$ este costul unui drum optim de la n la n’.

- Definiție 7.5: O euristică h este monotonă dacă pentru oricare două noduri n și n’ ale grafului explorat, astfel încât n’ este succesorul lui n, există inegalitatea $h(n) \leq h(n') + c(n,n')$, unde $c(n,n')$ este costul arcului (n,n').

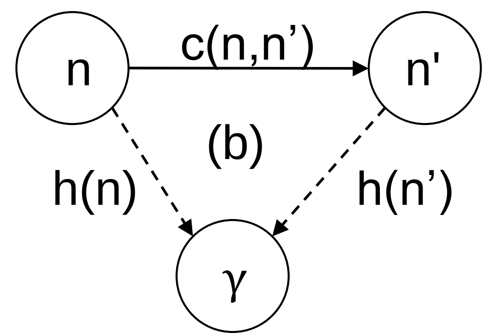


$h(n) \leq h(n') + k(n,n')$

Regula triunghiului pentru euristici:

← Consistență

→ Monotone



$h(n) \leq h(n') + c(n,n')$

Consistență = monotonie

- Teorema 7.5: O euristică este consistentă ⇔ este monotonă.
  - Demonstrație:
    - h – consistentă → h – monotonă. Alegem n' ∊ succs(n) → k(n,n') = c(n,n') → h(n) ≤ h(n') + c(n,n') → h – monotonă.
    - h – monotonă → h – consistentă. Fie $n = n_1,n_2,\ldots,n_q = n'$, un drum optim n..n' cu cost k(n,n'). → $h(n) = h(n_1) \leq h(n_2) + c(n_1,n_2) \leq h(n_3) + c(n_1,n_2) + c(n_2,n_3)\ldots \leq h(n_q) + c(n_1,n_2) + c(n_2,n_3) + \ldots c(n_{q-1},n_q) = h(n_q) + k(n_1,n_q)$ → $h(n) \leq h(n') + k(n,n')$ → h – consistentă.

Consistență → admisibilitate

- Teorema 7.6: O euristică consistentă este admisibilă.
  - Demonstrație:
    - Fie h o euristică consistentă → $h(n) \le h(n') + k(n,n')$, $\forall$ n' accesibil din n. Fie $n' = \gamma \in \Gamma$ → $k(n, \gamma) = \min \{ k(n, \gamma') \mid \gamma' \in \Gamma\} = h^*(n)$ → $h(n) \le h(\gamma) + h^*(n)$, dar $h(\gamma) = 0$ → $h(n) \le h^*(n)$ → euristică admisibilă.
- Corolar 7.2: O euristică monotonă este admisibilă.

Dominanță - Definiții și teoremă

- Definiție 7.6: Fie h_1 și h_2 două euristici admisibile. Spunem că h_1 este mai informată decât h_2 dacă $h_2(n) < h_1(n)$ pentru orice nod $n \notin \Gamma$ din graful spațiului de stare explorat.

- Definiție 7.7: Un algoritm A_1^* domină un algoritm A_2^* dacă orice nod expandat de A_1^* este expandat și de A_2^*. (eventual, A_2^* expandează noduri suplimentare față de A_1^*, deci A_1^* poate fi mai rapid ca A_2^*.)

- Teorema 7.11: Dacă o euristică monotonă h_1 este mai informată decât o euristică monotonă h_2, atunci un algoritm A_1^* condus de h_1 domină un algoritm A_2^* condus de h_2.

Dominanța - Exemplu

- Considerăm jocul 8-pătrățele care trebuie aranjat pornind de la forma inițială prin mutarea locului 'liber' astfel încât să ajungem la forma finală:

7	4	1
5	6	3
2	8	

1	2	3
4		5
6	7	8

- Două euristici posibile:
  - h_1 = numărul pătrățelelor a căror poziție curentă diferă de poziția finală;
  - $h_1 = \Sigma_{p \in piese}(\delta_p)$, unde $\delta_p = 0$ dacă poziția curentă coincide cu cea finală și $\delta_p = 1$, altfel
  - h_2 = distanța Manhattan = suma distanțelor pe verticală și orizontală între pozițiile curente ale pătrățelelor și pozițiile lor finale
  - $h_2 = \Sigma_{p \in piese}(dist\_h_p + dist\_v_p)$

Admisibilitate? Monotonie? Dominanță? Care euristică va fi aleasă pentru A*?

Complexitate A*

- Liniară dacă $|h^*(n) - h(n)| \le \delta$, unde $\delta \ge 0$ este o constantă.
- Subexponențială, dacă $|h^*(n) - h(n)| \le O(\log(h^*(n)))$.
- Exponențială, altfel, (dar mult mai bună decât a căutărilor neinformate).
- Mai multe explicații găsiți în Giumale 7.4.4!

Proiectarea Algoritmilor

Algoritmi aleatorii

Obiective

- Definirea conceptului de algoritm aleator
- Algoritmi Las Vegas
- Algoritmi Monte Carlo
- Analiza algoritmilor aleatori

Algoritmi aleatori

- Micșorăm timpul de rezolvare a problemei relaxând restricțiile impuse soluțiilor.
- Determinarea soluției optime:
  - Renunțăm la optimalitate (soluția suboptimală are o marjă de eroare garantată prin calcul probabilistic).
- Găsirea unei singure soluții:
  - Găsim o soluție ce se apropie cu o probabilitate măsurabilă de soluția exactă.

Algoritmi Las Vegas

- Găsesc soluția corectă a problemei, însă timpul de rezolvare nu poate fi determinat cu exactitate.
- Creșterea timpului de rezolvare → creșterea probabilității de terminare a algoritmului.
- Timp = ∞ → algoritmul se termină sigur (soluția e optimă).
- Probabilitatea de găsire a soluției crește extrem de repede astfel încât să se determine soluția corectă într-un timp suficient de scurt.

Algoritmi Monte Carlo

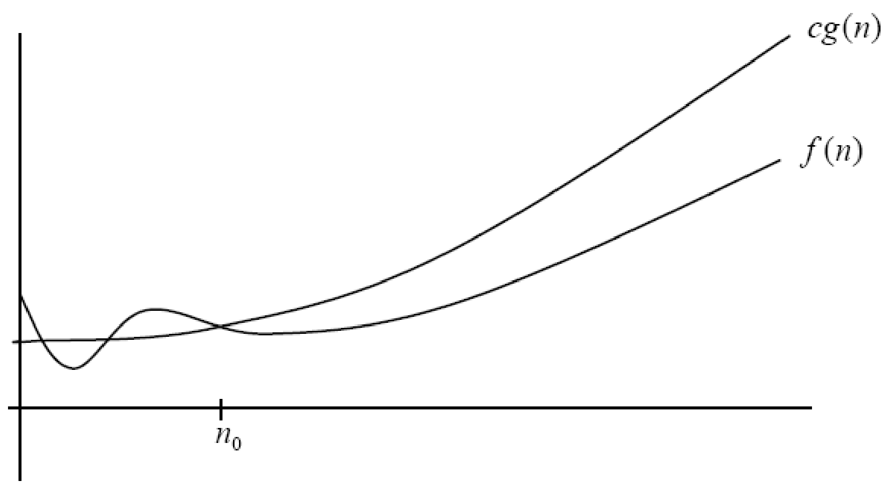
- Găsesc o soluție a problemei care nu e garantat corectă (soluție aproximativă).
- Timp = ∞ → soluția corectă a problemei.
- Probabilitatea ca soluția să fie corectă crește o dată cu timpul de rezolvare.
- Soluția găsită într-un timp acceptabil este aproape sigur corectă.

Reminder – complexitatea algoritmilor

$$O(g(n)) = \{ f(n) \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) \leq cg(n) \}$$

- g(n) - limita asimptotică superioară pentru f(n)

$f(n) = O(g(n))$ says:



http://classes.soe.ucsc.edu/cmps102/Spring04/TantaloAsymp.pdf

Complexitatea algoritmilor Las Vegas

- Definiția 6.1: Algoritmii Las Vegas au complexitatea $f(n) = O(g(n))$ dacă $\exists$ $c > 0$ și $n_0 > 0$ a.î. $\forall$ $n \geq n_0$ avem $0 < f(n) < c\,\alpha\,g(n)$ cu o probabilitate de cel puțin $1 - n^{-\alpha}$ pentru $\alpha > 0$ fixat și suficient de mare.

Complexitate algoritmi Monte Carlo

- Definiția 6.1’: Algoritmii Monte Carlo au complexitatea $f(n) = O(g(n))$ dacă $\exists\ c > 0$ și $n_0 > 0$ astfel încât:
  - $\forall n \geq n_0,\ 0 < f(n) \leq c\ \alpha\ g(n)$ cu o probabilitate de cel puțin $1 - n^{-\alpha}$ pentru $\alpha > 0$ fixat si suficient de mare;
  - Probabilitatea ca soluția determinată de algoritm să fie corectă este cel puțin $1 - n^{-\alpha}$.

Exemplu algoritm Las Vegas

- 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;
  - 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 10% din numărul de linii al secvenței.
  - Secvențele sunt lungi.☺

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

Rezolvare “clasică”

Detectează_linii(fișier)

- **Pentru fiecare** Secv ∈ fișier
  - **Pentru** i **de la** 0 **la** dim(Secv)
    - **Pentru** j **de la** i + 1 **la** dim(Secv)
      - **Dacă** linie(i,Secv) = linie(j,Secv) **atunci**
        - **Întoarce** (linie(i,Secv));

prelucrare secvență

Complexitate – O(dim(Secv)^2)

Algoritm Las Vegas pentru rezolvarea problemei

- Secțiunea “prelucrare secvență” se înlocuiește cu următoarea funcție:

- Selecție_linii(n,secv) // n = dim secv
  - **Cât timp**(1) // mereu
    - i = random(0,n-1) // selectez o linie
    - j = random(0,n-1) // și încă una
    - **Dacă** i != j **și** linie(i,Secv) = linie(j,Secv) **atunci** // le compar
      - **Întoarce** linie(i,Secv) // am găsit linia

Analiza algoritmului Las Vegas (I)

- Notații:
  - n = numărul de linii din secvența curentă;
  - q = ponderea liniei repetate în secvență;
  - r = numărul de apariții al liniei repetate: r = n * q / 100;
  - m = numărul de pași necesari terminării algoritmului;
  - P_k = probabilitatea ca la pasul k să fie satisfăcută condiția de terminare a algoritmului;
  - P(m) = probabilitatea ca algoritmul să se termine după m pași.

Analiza algoritmului Las Vegas (II)

- Probabilitatea ca la pasul k linia i să fie una din liniile repetate este r / n.
- Probabilitatea ca la pasul k linia j să fie una din liniile repetate (diferită de i) este (r - 1) / n.
- Condiția de terminare: cele 2 evenimente trebuie să se producă simultan:

$$P_k = r / n * (r-1) / n = q / 100 * (q / 100 – 1 / n)$$

Analiza algoritmului Las Vegas (III)

- Probabilitatea ca algoritmul să NU se termine după m pași:
  - $\Pi_{k = 1 \rightarrow m}(1 - P_k) = \Pi_{k = 1 \rightarrow m} [1 – q / 100 * (q / 100 – 1 / n)] = [1 – q / 100 * (q / 100 – 1 / n)]^m$
- $\rightarrow$ $P(m) = 1 - [1 – q / 100 * (q / 100 – 1 / n)]^m$
- Pp: n > 100; q > 10%
- $\rightarrow$ $P(m) \geq 1 – [1 – q * (q – 1) / 10.000]^m$

Comparație timp de rulare

- q = 10%:
  - 3500 pași P = 1;
  - 1000 pași – P = 0,9988.
- q = 20%:
  - 1000 pași P = 1.
- q = 30%:
  - 500 pași P = 1.
- Varianta clasică: cazul cel mai defavorabil – 10000 pași.

Complexitate algoritm Las Vegas

- Algoritmii Las Vegas au complexitatea $f(n) = O(g(n))$ dacă $\exists$ $c > 0$ si $n_0 > 0$ a.i. $\forall$ $n \geq n_0$ avem $0 < f(n) < c\ \alpha\ g(n)$ cu o probabilitate de cel puțin $1 - n^{-\alpha}$ pentru $\alpha > 0$ fixat si suficient de mare.

- Arătăm că $f(n) = O(lg(n))$:
  - Notăm: $a = 1 – q * (q - 1) / 10.000$;
  - $1 – P(c\ \alpha\ lg(n))$ = probabilitatea ca algoritmul să nu se termine în $c\ \alpha\ lg(n)$ pași;
  - $P(c\ \alpha\ lg(n)) \geq 1- a^{c\ \alpha\ lg(n)} \rightarrow 1 – P(c\ \alpha\ lg(n)) \leq a^{c\ \alpha\ lg(n)} = n^{c\ \alpha\ lg(a)} = n^{-c\ \alpha\ lg(1/a)}$ pentru că $0 < a < 1$;
  - Dacă alegem $c \geq lg^{-1}(1/a) \rightarrow 1 – P(c\ \alpha\ lg(n)) \leq n^{-\alpha} \rightarrow P(c\ \alpha\ lg(n)) \geq 1 - n^{-\alpha} \rightarrow$ algoritmul se termină în $lg^{-1}(1/a)\ \alpha\ lg(n)$ pași cu o probabilitate $\geq 1 - n^{-\alpha} \rightarrow$ (definiție) $f(n) = O(lg(n))$.

PA

Exemplu algoritm Monte Carlo

- Problemă: testarea dacă un număr n dat este prim.
- Rezolvare "clasică":

Complexitate: O(sqrt(n))

- Prim-clasic(n)
  - **Pentru** i **de la** 2 **la** sqrt(n)
    - **Dacă** (n mod i == 0) **întoarce** fals;
  - **Întoarce** adevărat

Determinarea numerelor prime - complexitate

- Observație: pentru numere mari – operațiile nu mai durează O(1)!
- → Estimăm numărul de operații în funcție de numărul de biți pe care este exprimat numărul.
- → Prim_clasic – $O(2^{k/2})$ unde k = nr. de biți ocupat de n.

Complexitate nesatisfăcătoare!

- "On September 4, 2006, in the same room just a few feet away from their last find, Dr. Curtis Cooper and Dr. Steven Boone's CMSU team broke their own world record, discovering the 44th known Mersenne prime, $2^{32,582,657}-1$. The new prime at 9,808,358 digits is 650,000 digits larger than their previous record prime found last December."

- "On April 12th (2009) , the 46th known Mersenne prime, $2^{42,643,801}-1$, a 12,837,064 digit number was found by Odd Magnar Strindmo from Melhus, Norway! This prime is the second largest known prime number, a "mere" 141,125 digits smaller than the Mersenne prime found last August."

- As of October 2009, 47 Mersenne primes are known. The largest known prime number ($2^{43,112,609} - 1$) is a Mersenne prime. [Wikipedia]

http://www.mersenne.org

Algoritm aleator (I)

- Teorema 6.1 (mica teoremă a lui Fermat): Dacă n este prim → $\forall\ 0 < x < n, x^{n-1} \bmod n = 1$.

- Prim1(n,α) // detectează dacă n e număr prim
  - **Dacă** (n ≤ 1 **sau** n mod 2 = 0) **Întoarce** fals
  - Limit = limită_calcul(n,α) // numărul minim de pași pentru
    // soluția corectă cu $P = 1 - n^{-\alpha}$
  - **Pentru** i **de la** 0 **la** limit
    - x = random(1, n-1) // aleg un număr oarecare
    - **Dacă** (pow_mod(x,n) ! = 1) **Întoarce** fals // testez teorema
      // Fermat
  - **Întoarce** adevărat

Complexitate?

Algoritm aleator (II)

- Pow_mod(x,n) // calculează x^{n-1} mod n
  - r = 1 // restul
  - **Pentru** m **de la** n-1 **la** 0
    - **Dacă** (m mod 2 ≠ 0) // testez dacă puterea e pară // sau nu
      - r = x * r mod n
    - x = (x * x) mod n // calculez x^2 mod n
    - m = m div 2 // înjumătățesc puterea
  - **Întoarce** r

Complexitate: O(lg(n))

Algoritm aleator (III)

- Problemă: nu putem stabili cu exactitate care este limita de calcul:
  - Nu se poate estima pentru un număr compus n numărul de numere x, 2 < x < n pentru care nu se verifică ecuația;
  - Există numere compuse pentru care orice număr x < n și prim în raport cu n satisface ecuația lui Fermat (ex: nr. Carmichael → 561).
- → Nu știm cu exactitate câte numere sunt!
  → Nu putem calcula probabilitatea!

Altă variantă de algoritm aleator

- Teorema 6.2: Pentru orice număr prim n ecuația x^2 mod n = 1 are exact 2 soluții:

  $x_1 = 1$ ȘI $x_2 = n – 1$.

- Definiție 6.2: Fie n > 1 și 0 < x < n două numere astfel încât x^{n-1} mod n $\neq$ 1 sau x^2 mod n = 1, x $\neq$ 1 și x $\neq$ n – 1. X se numește martor al divizibilității lui n.

Algoritmul Miller-Rabin

- Prim2(n,α)
  - **Dacă** (n ≤ 1 **sau** n mod 2 = 0) **Întoarce** fals
  - limit = limita_calcul(n,α)
  - **Pentru** i **de la** 0 **la** limit
    - x = random(1,n-1)
    - **Dacă** (martor_div(x,n)) **Întoarce** fals
  - **Întoarce** adevărat

Complexitate?

Algoritmul Miller-Rabin (II)

- martor_div(x,n) // determină dacă x e
  // martor al divizibilității lui n
  - r = 1; y = x;
  - **Pentru** m **de la** n-1 **la** 0 // puterea
    - **Dacă** (m mod 2 $\neq$ 0) // putere impară
      - r = y * r mod n
    - z = y // salvez valoarea lui x
    - y = y * y mod n // calculez y^2 mod n
    - **Dacă** (y = 1 **și** z $\neq$ 1 **și** z $\neq$ n-1) // verific teorema 6.2
      - **Întoarce** 1
    - m = m div 2 // înjumătățesc puterea
  - **Întoarce** r $\neq$ 1 // mica teoremă Fermat (x^{n-1} mod n $\neq$ 1)

Complexitate:
O(lg(n))

Calcularea numărului de pași

- Teorema 6.3: Pentru orice număr n, impar și compus există cel puțin (n-1) / 2 martori ai divizibilității lui n.
- Caz neinteresant: număr prim pentru că oricum algoritmul întoarce adevărat ($P_{corect}(n) = 1$)!
- Caz interesant: număr compus (impar) ($P_{corect}(n) = ?$):
- x = element generat la un pas al algoritmului ($0 < x < n$);
- P(x) = probabilitatea ca numărul **x** generat din cele n-1 posibilități să fie martor al divizibilității;
- $P(x) \geq (n-1) / 2 * 1 / (n-1) = 0.5$;
- $P_{incorect}(n) = \Pi_{1\text{->}limit}(1 - P(x)) \leq 1/2^{limit}$;
- → $P_{corect}(n) \geq 1-2^{-limit} = 1 - n^{-\alpha}$ → limit = α lg(n); →după α lg(n) pași $P_{corect}(n) \geq 1 - n^{-\alpha}$;
- → Complexitate: $O(lg^2(n))$ → în funcție de numărul de biți k → Complexitate: $O(k^2)$

Exemplu de utilizare practică

- Quicksort(A, st, dr)
  - **Dacă** st < dr
    - q ← Partiție(A, st, dr)
    - Quicksort(A, st, q - 1)
    - Quicksort(A, q + 1, dr)

Cazul defavorabil?

- Partiție(A, st, dr)
  - x ← A[dr]
  - i ← st - 1
  - **Pentru** j **de la** st **la** dr – 1
    - **Dacă** A[j] ≤ x
      - i ← i + 1
      - **Interschimbă** A[i] ↔ A[j]
  - **Interschimbă** A[i + 1] ↔ A[dr]
  - **Întoarce** i + 1

Complexitate?

Exemplu de utilizare practică (II)

- Problema Quicksort – cazul defavorabil – datele de intrare sunt sortate în ordine inversă.
- Complexitate Quicksort: $O(n^2)$.
- Folosind algoritmi aleatori eliminăm acest caz.

Quicksort-aleator

- Quicksort-Randomizat(A, st, dr)
  - **Dacă** st < dr
    - q ← Partiție-Randomizată(A, st, dr)
    - Quicksort-Randomizat(A, st, q - 1)
    - Quicksort-Randomizat(A, q + 1, dr)

- Partiție-Randomizată(A, st, dr)
  - i ← Random(st, dr)
  - **Interschimbă** A[dr] ↔ A[i]
  - **Întoarce** Partiție(A, st, dr)

INTREBĂRI?