Lowest Common Ancestor

- Analiza Algoritmilor -

Philip DUMITRU
324CD
Facultatea de Automatică şi Calculatoare
Universitatea Politehnică din Bucureşti
ph.dumitru@gmail.com

December 14, 2017

Abstract

Lucrarea conţine abordarea problemei găsirii celui mai mic strămoş comun al unor noduri dintr-un arbore.

1 Introducere

1.1 Importanţa găsirii Lowest Common Ancestor[1]

Cea mai intâlnită structură din lume este structura arborescentă. Ea este prezentă în toate domenile, de la fractali, botanica, medicina, până la reţelistica, adminitraţie, economie...

De multe ori în reţelistică, pentru a trimite pachete între 2 device-uri, trebuie să ştim care este drumul direct între acestea. Este evident, datorită proprietăţilor arborilor, că acest drum va trece printr-un device intermediar ce reprezintă strămoşul comun a nodurilor reprezentate de device-uri.

În genetică, pentru a găsi individul ce a răspândit o anumită mutaţie, trebuie să aflăm care este strămoşul comun al tuturor indivizilor ce prezintă mutatia.

Un alt exemplu este in controlul versiunilor. Pentru a determina automat modul în care 2 fişiere trebuiesc unite(merged), se determină mai întâi fişierul din care derivă ambele. Algoritmul Three-way merge[2] foloseşte această abordare.

1.2 Enunţarea problemei

Aşadar, problema ce va fi expusă în acest articol este următoarea:

> Se dă un arbore T cu N noduri şi apoi se fac M interogări astfel: pentru fiecare interogare se dau 2 noduri u şi v. Trebuie să returnaţi nodul w cu cea mai mare adâncime din T care este strămoş atât pentru u, cât şi pentru v.

1.3 Despre algoritmii de calculare LCA

Precum se poate vedea, calcularea LCA este una dintre cele mai importante operaţii efectuate pe arbori. Exista multiplii algoritmi, de complexităţi diferite, ce efectuează acest calcul. Performanţele acestora însă depind de structurarea arborelui, precum şi de numărul de astfel de operaţii ce se efectuează.

Algoritmii descrişi vor fi următorii: Metoda Brute-force($\mathcal{O}(M * N)$), Algoritmul offline al lui Tarjan ($\mathcal{O}(N\log N + M)$), Algoritmul ce foloseste Range Minimum Query[3] ($\mathcal{O}(N\log N + M)$), precum şi un algoritm de complexitate $\mathcal{O}(N\log N + M\log N)$.

[1] îl vom nota cu LCA
[2] vezi wikipedia
[3] îl vom nota cu RMQ

2 Descrierea şi implementarea algoritmilor

2.1 Arborele

Pentru a pune problema calculării LCA a 2 noduri, trebuie mai întai considerată o structură de tip arbore. Această structură presupune încapsularea fiecărei entităţi într-un nod. Fiecare dintre acestea va avea un părinte şi o listă de fii, excepţia fiind nodul rădăcină ce nu are niciun părinte. Deasemenea, între oricare 2 noduri ale arborelui se poate indentifica un lanţ de părinţi sau fii.

Cel mai simplu nod ce se poate crea pentru a încapsula datele efective este o structură ce conţine:

- o legătura către părinte
- o listă de legături către copii
- un index pentru identificarea nodului
- datele corespunzătoare nodului

În acest mod se poate abstractiza orice structură de informaţii, permiţand aplicarea unui algoritm de determinare a LCA, întrucât primele 3 elemente ale nodurilor conţin toate informătile necesare despre arbore.

Pentru exemplificarea Algoritmilor prezentaţi mai jos, împreună cu acest document, se oferă o implementare a unui arbore simplu ce poate încapsula orice tip de date. În exemplu se folosesc chiar indicii pe post de tip de date. Arborele este implementat în fişierul *src\tree.h*

2.2 BruteForce

Această metodă constă în abordarea directă a căutării strămoşului, fāra a face niciun fel de pre-procesare. Într-un mod simplist, se efectuează două parcurgeri ale arborelui dinspre frunze spre rădăcină, începdând cu nodurile u respectiv v. Primul nod comun în cele 2 parcurgeri va fi chiar nodul w reprezentând LCA pentru u şi v. Din acest motiv, nu este necesar ca aceste 2 parcurgeri să fie complete, şi se pot opri la găsirea nodului comun.

Aşadar algoritmul bruteforce este următorul:

```
Function LCA(nod u, nod v)
    input  : u, v: nodurile cărora se calculează LCA
    output: cel mai mic strămoş al celor 2 noduri
    // aduce u si v la aceeasi înălţime
    while u.inaltime() > v.inaltime() do
        u = parinte(u)
    end
    while v.inaltime() > u.inaltime() do
        v = parinte(v)
    end
    // parcurge arborele în sus până la întâlnirea primului strămoş comun.
    while u ≠ v do
        v = parinte(v) u = parinte(u)
    end
    return v
```

Algorithm 1: LCA bruteforce

Implementarea acestui algoritm se găseşte în fişierul *src\AlgorithmLCA.h*

2.3 Algoritmul offline al lui Tarjan pentru LCA

Acest algoritm se foloseşte de structura de date mulţimi disjuncte, precum şi de operaţiile pe arbori Union şi Find. Utiliatea folosirii mulţimilor este identificarea rapidă a apartenenţei unui nod la un subarbore. Dezavantajul acestui algoritm este necesitatea cunoaşterii interogărilor inainte de efectuarea preprocesării, ceea ce inseamnă folosirea unei memorii adiţionale proporionala cu M. Mai mult, interogările nu vor fi parcurse în ordinea dată, ci în funcţie de ordinea parcurgerii nodurilor. In caz necesar, interogările pot fi reordonate în urma execuţiei algoritmului.

```
Function TarjanLCA(nod u)
    input : u, v: nodurile cărora se calculează LCA
    perechiUV: listă cu perechi de noduri u, v
    output: cel mai mic strămoş al celor 2 noduri

    // crează un nou set
    MakeSet(u)
    u.strămoş = u
    foreach c in u.copii do
        TarjanLCA(c) // apelează recursiv LCA în arbore
        Union(u, c) // uneşte setul tocmai creat cu cel al părintelui
        Find(u).strămoş = u // modifică strămoşul
    end
    u.marcat = true
    foreach (u, v) in perechiUV do
        if v.marcat == true then
            LCA(u, v) = Find(v)
        end
    end
```

Algorithm 2: LCA Tarjan

Algoritmul realizează la fiecare pas câte o mulţime pentru fiecare fiu parcurs al său, ce conţine toate nodurile ce au ca strămoş comun acel fiu. Se parcurge lista de interogări, iar dacă se găsesc ambele noduri u şi v în submulţime, atunci LCA va fi considerat ca fiind chiar nodul curent. Deoacece arborele este parcurs în postordine, întotdeauna se garantează că toate nodurile parcurse sunt urmaşi ai acestui nod. Ca urmare, se poate aplica recursiv acest algoritm asupra acestei mulţimi, pentru rezolvarea interogărilor ce au ambele noduri în subarborele fiului.

În momentul când o interogare a fost rezolvată, aceasta nu va mai fi recalculată ulterior. Asadar, dat fiind un nod t, cu având cel puţin 2 fii, $left$ şi $right$, precum şi o interogare (u, v), se pot întâlni următoarele cazuri:

1. u şi v ambele aparţin subarborelui aceluiaşi fiu, deci interogarea (u, v) a fost rezolvată la pasul corespunzător fiului

2. u şi v aparţin unor subarbori determinaţi de fii diferiti($left$ respectiv $right$), sau unul din noduri este chiar nodul curent. În acest caz se garantează că nodul curent este LCA pentru cele două noduri

3. u aparţine unui subarbore determinat de unul dintre fii, însă v este situat într-un loc total diferit în arbore). Această interogare va fi sărită la pasul curent, fiind necesară rezolvarea sa ulterior

Tratând cazurile astfel, rezolvarea unei interogări va fi întotdeauna nodul curent, dacă interogarea nu a fost înca analizată si poate fi rezolvată.

Implementarea acestui algoritm se găseşte în fişierul *src\TarjanOfflineLCA.h*

Mulţimi. Union-Find :

Operaţile pe mulţimi necesare algoritmului Tarjan sunt *MakeSet*, ce crează o nouă mulţime cu un singur element, *Union*, ce uneşte 2 mulţimi, şi *Find* ce identifică mulţimea din care face parte un element.

2.4 Algoritmul RMQ pentru LCA

Soluţia aceasta propune reducerea problemei găsirii strămoşului comun la problema găsirii minimului unei subsecvenţe a unui şir. Transformarea datelor de intrare se realizează prin aplatizarea arborelui folosind parcurgerea euleriană[4]. Şirul obţinut va fi format din indici ai nodurilor, iar LCA a 2 noduri u şi v va fi dat de indicele nodului de înălţime minimă între o apariţie a indicelui nodului u şi o apariţie a indicelui nodului v.

Pentru rezolvarea problemei RMQ, vom folosi programarea dinamică, de dimensiune $N \times \log N$. Fiecare element (i, j) va conţine minimul dintre 2^j elemente începând cu indicele i. Se poate astfel calcula minimul dintre poziţile x şi y prin calculul minimului dintre dinamic(x, $\log x-y$) şi dinamic(”$y-2^{[\log(x-y)]}$, $\log(x-y)$).

```
Function RMQ_Preprocesare()
    input : arbore: structura în care se caută LCA
    output: Tabel: dinamica aplicată şirului
    Aplatizează(arbore)
    // arbore este acum un şir
    for j ← 0 to log(arbore.size) do
        for i ← 1 to arbore.size do
            Tabel(i, j) = ÎnalţimeMimină(Tabel(i, j − 1), Tabel(i + 2^(j−1), j − 1))
        end
    end
```

Algorithm 3: LCA RMQ preprocesare

```
Function RMQ_LCA(nod u, nod v)
    input : u, v: nodurile cărora se calculează LCA
    output: cel mai mic strămoş al celor 2 noduri
    // Găseşte o poziţie a nodurilor în şirul arborelui aplatizat
    i = IndiceÎnŞir(u) j = IndiceÎnŞir(v)
    if i > j then i ⟷ j
    return ÎnalţimeMimină(Tabel(i, [log(j − i + 1)]), Tabel(j − 2^[log(j−i+1)] + 1, [log(j − i + 1)]))
```

Algorithm 4: LCA RMQ

Implementarea acestui algoritm se găseşte în fişierul *src\RMQ_LCA.h*

Această reducere a problemei LCA la RMQ, ne dă posibilitatea aplicării şi altor algoritmi precum rezolvarea cu arbori de intervale.

Euler tour tree :

Parcurgerea euleriană a unui arbore se aseamănă cu parcurgerile clasice ale arborilor. În timp ce parcurgerea preordine vizitează mai întâi părintele, apoi fiecare copil, iar parcurgerea postordine mai intai copii, si apoi părintele, parcurgerea euleriană este o combinatie a acestora. Mai mult, această parcurgere vizitează părintele de mai multe ori: odată la începutul parcurgerii, odată după vizitarea tuturor copiilor, precum şi între parcurgerea copiilor.

Ca exemplu, putem considera arborele cu rădăcina în **root**, ce are copiii **left**, **middle** şi **right**. Parcurgerea euleriană este: **root**, **left**, **root**, **middle**, **root**, **right**, **root**.

Relevanţa acestei parcurgeri pentru problema găsirii LCA este faptul că pentru 2 noduri u şi v, LCA va fi vizitat între parcurgerea subarborelui cu rădăcina în u şi parcurgerea subarborelui cu rădăcina în v. De reţinut este că dimensiunea şirului aplatizat este aproximativ dublul numărului de noduri.

2.5 Parcurgerea arborelui în timp logaritmic

Deşi acest algoritm este mai puţin eficient decât algoritmul Tarjan sau RMQ, el este unul destul de intuitiv, ce deasemenea se implementează rapid.

Asemănator cu metoda Bruteforce, strămoşul se caută în arbore de la frunze spre rădăcină. Diferenţa însă constă în numărul paşilor de parcurgere. În timp ce Bruteforce trecea prin fiecare nod situat între

[4] Euler tour tree

u, v şi w, acest algoritm parcurge numai anumite noduri aflate la diferite distanţe. Acestea se determină printr-o dinamică similară cu cea de la RMQ, dar aplicată direct asupra arborelui.

Dinamica folosită va fi de dimensiune $N \times \log N$. Fiecare element (i, j) va conţine indicele strămoşului de ordin j al lui i (notăm i.strămoş(j)), unde i.strămoş(j) se poate scrie recurent ca fiind (i.strămoş($j-1$)).strămoş($j-1$). Evident, i.strămoş(0) este părintele nodului i. Aşadar algoritmul este următorul:

```
Function Log_Preprocesare()
    input  : arborele
    output: strămoş: dinamica aplicată şirului

    for j ← 1 to log(arbore.size) do
        for i ← 1 to arbore.size do
            i.strămoş(j) = (i.strămoş(j − 1)).strămoş(j − 1)
        end
    end
```

Algorithm 5: LCA în timp logaritmic - preprocesare

```
Function Log_LCA(nod u, nod v)
    input  : u, v: nodurile cărora se calculează LCA
    output: cel mai mic strămoş al celor 2 noduri

    if u.inaltime() < v.inaltime() then u ⟷ v

    // aduce u si v la aceeasi înălţime
    for i ← log(.size) to 0 do
        if u.inaltime() - 2^i ≥ v.inaltime() then
        u = u.strămoş(i)
    end

    // parcurge arborele în sus până la întâlnirea primului strămoş comun.
    for i ← log(.size) to 0 do
        if u.strămoş(i) ≠ v.strămoş(i) then
        u = u.strămoş(i)
        v = v.strămoş(i)
    end
    if u == v then
        return u
    else
        return u.parinte
    end
```

Algorithm 6: LCA în timp logaritmic

Implementarea acestui algoritm se găseşte în fişierul *src\Log_LCA.h*

3 Complexitate temporală

3.1 Dimensiunile problemei

Pentru a discuta complexitatea rezolvării problemei LCA, este nevoie de a identifica dimensiunea datelor de intrare.

Se pot identifica imediat parametrii N (numărul de noduri) şi M (numărul de interogări). Totusi, datorită naturii problemei, determinarea LCA va fi influenţată şi de lungimea lanţului dintre 2 noduri, deci de configuratia arborelui. Întrucât distanţa maximă dintre 2 elemente ale unui arbore este mai mică decât dublul înăltimii arborelui, putem considera H (înaltimea arborelui) ca fiind parametrul ce descrie configuratia arborelui. De reţinut că H depinde de N.

Vom efectua deasemenea şi următoarele convenţii: vom numi un arbore dens un arbore cu $H \ll \log N$ şi vom numi un arbore înalt un arbore cu $H \gg \log N$. Se poate observa că arborii denşi vor avea lanţuri mult mai mici între noduri decât arborii înalţi. Deasemenea, pe cazul mediu, $H \simeq \log N$.

3.2 BruteForce

Algoritmul Bruteforce necesită doar două parcurgeri în sus a arborelui pentru fiecare interogare, ceea ce înseamnă $\theta(H)$ fiecare interogare. Dacă pe lângă interogări considerăm pasul de preprocesare de complexitate $\theta(N)$ (datorată citirii şi construirii arborelui), se obţine complexitatea teoretică a algoritmului $\theta(N + M \cdot H)$. Dar întrucăt H depinde de configuraţia arborelui, cel mai rău caz va fi atunci cănd $H \simeq N$, deci complexitatea finală este $\mathcal{O}(M \cdot N)$

3.3 Algoritmul offline al lui Tarjan pentru LCA

Complexitatea algoritmului Tarjan depinde de implementarea operaţilor cu mulţimi Union şi Find. Cea mai eficientă implementare a acesora are complexitatea $\mathcal{O}((N))$ (unde (x) este inversul functii Ackermann), dar pentru simplitate, vom considera complexitatea $\mathcal{O}(\log N)$. $((x) < \log(x))$

Algoritmul se poate împărţi în 2 secţiuni: Parcurgerea arborelui şi verificarea strămoşului. Parcurgerea este una de tip DFS, având complexitate liniară. În cadrul acesteia se execută operaţia Union[5] pentru fiecare nod. Deci parcurgerea va avea complexitatea $\mathcal{O}(N \log N)$ Verificarea strămoşului are complexitatea $\mathcal{O}(M)$. Aceasta este dată de parcurgerea tuturor perechilor de noduri (u, v) de 2 ori (odată la vizitarea nodului u, odată la vizitarea nodului v).

Adăugând complexitatea celor 2 secţiuni, precum şi complexitatea preprocesării $\theta(N+M)$ (citirea şi construirea arborelui şi a perechilor (u, v)), algoritmul Tarjan se încadrează în complexitatea $\mathcal{O}(N \log N + M)$. [6]

3.4 Algoritmul RMQ pentru LCA

Complexitatea algoritmului RMQ implementat este dată de preprocesarea arborelui aplatizat. Aceasta constă în construirea în mod recurent al unei dinamici de dimensiune $N \times \log N$, şi este cea ce dă complexitatea preprocesării. Restul operaţilor, atât citirea şi construirea arborelui, căt si aplatizarea(prin parcurgere DFS), au timp liniar.

Rezolvarea unei interogări pentru algoritmul RMQ se face practic instant. Complexitatea este $\theta(1)$. Selectarea indicelui corespunzător poziţiei în arbore al unui nod se face şi ea în timp constant, prin reţinerea sa într-un vector la momentul preprocesării.

În final, se obţine complexitatea $\theta(N \log N + M)$ pentru aceaşta implementare RMQ.

Pentru referinţă, complexitatea algoritmului RMQ implementat cu arbori de intervale se obţine $\theta(N \log N + M)$.

3.5 Parcurgerea arborelui în timp logaritmic

Precum în cazul RMQ, preprocesarea va avea complexitatea $\theta(N \log H)$ (dinamică de aceeasi dimensiune). Interogările însă, vor avea complexitatea $\theta(\log H)$. Acest lucru se datorează parcurgerii arborelui în sus (de unde vine H), sărind exponenţial noduri (de unde provine log.

Însumand complexitătile se obţine $\theta(N \log H + M \log H)$, iar pentru arbori înalţi $\theta(N \log N + M \log N)$

4 Complexitate spatială

4.1 BruteForce

Ca majoriatea metodelor bruteforce, acesta foloseşte o memorie adiţionala de dimensiune $\theta(1)$.

[5] Ca fapt divers, aceaşta operaţie se comportă mai bine pe arbori denşi

[6] Complexitatea reală este $\mathcal{O}(N\alpha(N) + M)$ şi poate fi aproximată chiar cu $\mathcal{O}(N + M)$ deoarece $\alpha(N) < 5$ chîar şi pentru valori extrem de mari

4.2 Algoritmul offline al lui Tarjan pentru LCA

Aşa cum am precizat de la început, pentru aplicarea algoritmului lui Tarjan este nevoie cunoasterea în preambul al interogărilor. Acestea se vor stoca în memorie, ocupănd un spaţiu de dimensiune M. În plus, mai este nevoie de o memorie de dimensiune N pentru reţinerea mulţimilor create. Nu în ultimul rănd, pentru a putea fi executat algoritmul, este nevoie de un spatiu de dimensiune H pe stivă pentru apelurile recursive al parcurgerii DFS. Se obţine o memorie aditionala $\theta(M + N)$

4.3 Algoritmul RMQ pentru LCA

Implementarea algoritmului RMQ necesită, pentru dinamică o memorie de dimensiune $\theta(N \log N)$. Deasemenea se foloseşte şi o memorie liniară pentru reţinerea arborelui aplatizat.

4.4 Parcurgerea arborelui în timp logaritmic

Precum şi in cazul timpului, algoritmul de parcurgere a arborelui în timp logaritmic se comportă asemănator cu RMQ, necesitănd o memorie $\theta(N \log N)$.

5 Concluzie

5.1 Evaluarea algoritmlor pe un set de teste

Pentru a verifica eficienţa algoritmilor în practică, am realizat generatori de arbori cu care am realizat 70 de fisiere de intrare. Aceste fişiere se împart astfel:

1. 01..30: arbori denşi de diferite dimensiuni, cu un număr proporţional de interogări
2. 31..40: arbori denşi de dimensiuni mari, multe interogări
3. 41..50: arbori înalţi de dimensiuni medii, cu un număr apropiat de interogări
4. 51..60: arbori înalţi de dimensiune mare, dar puţine interogări
5. 61..70: arbori înalţi de dimensiune mică, dar multe interogări

Rezultatele au fost apropiate de cele asteptate[7] : pe arbori denşi, cel mai bine s-a comportat Tarjan ($\mathcal{O}(N \log N + M)$), iar apoi metoda Bruteforce cu complexitatea teoretică $\mathcal{O}(N + M \cdot N)$.

Acest lucru nu este surprinzător, deoarece complexitata sa depindea mai mult de H decât de N ($\theta(N + M \cdot H)$). Cel mai încet algoritm a fost RMQ, fiind singurul al cărei complexitate nu depinde în niciun fel de înaltimea arborelui.

Continuând cu arbori înalţi, ce reprezintă worst case, algoritmul Tarjan înca este foarte eficient. Pentru testele cu puţine interogări, este cel mai eficient algoritm, iar pe celelalte, este intrecut doar de algoritmul ce foloseşte RMQ. Desigur, cel mai slab algoritm este cel ce aplică metoda bruteforce.

5.2 Analiza rezultatelor în funcţie de parametrii

($N \gg M$, $H \gg \log N$) Dacă arborele este unul înalt, iar numărul nodurilor este mult mai mare decăt numărul de interogări, toţi algoritmii se vor comporta în mod asemănator, mai puţin metoda bruteforce. Complexitatea lor va fi aproximativ $\mathcal{O}(N \log N)$. În acest caz însa, există o rezolvare ce se comportă mai bine decât oricare din algoritmii prezentaţi aici. Este vorba de implementarea algoritmului RMQ cu arbori de intervale, ce are complexitatea $\mathcal{O}(N + M \log N)$.

($N \simeq M$ sau $N \ll M$, $H \gg \log N$) Dacă arborele este unul înalt, iar numărul nodurilor este mai mic decât cu numărul de interogări, cele mai bune rezultate se vor obţine pentru Tarjan şi RMQ, ambele având complexitatea $\mathcal{O}(N \log N + M)$. În acest caz, parcurgerea logaritmică este mai lentă, deoarece coeficientul logaritmic al lui N devine relevant ($\theta(N \log H + M \log H)$).

[7] rezultatele se pot vizualiza în fişierele time_file.algoritm.txt, ce se generează la rularea checrului in LINUX

($H \ll \log N$) Dacă arborele este unul dens, iar numărul nodurilor este mai mare decăt numărul de interogări, complexilatea algoritmilor bruteforce ($\theta(N + M \cdot H)$) şi parcugere logaritmică ($\theta(N \log H + M \log H)$) este mult mai mică decăt cea teoretică. Acestea ajung chiar să întreaca performanţele implementării algoritmului RMQ. Dacă ar fi să analizăm şi implementarea RMQ folosind arbori de intervale, acesta nu ar aduce nicio îmbunătătire problemei, fiind deasemenea întrecută.

5.3 Alegerea algoritmului

Deşi algoritmul lui Tarjan s-a dovedit a fi cel mai rapid în toate cazurile, există situaţii în care se poate prefera alt algoritm. De exemplu pentru cazul $N \gg M$, $H \gg \log N$, este mai eficient folosirea RMQ cu arbori de intervale. Altă situaţie ar fi când fiecare interogare depinde de o alta anterioară, caz în care algoritmul Tarjan, ce rezolvă offline problema LCA, nu mai funcţionează.

Un alt exemplu interesant este acela în care arborele se schimbă în mod dinamic. Acest lucru poate duce la imposibilitatea preprocesării datelor. În acest caz, algoritmul de parcurgere în timp logaritmic are o dinamică uşor modificabilă, introducerea unui nod nod fiind echivalentă cu introducerea unei singure coloane în dinamica folosită.

Se poate concluziona că alegerea algoritmului depinde în totalitate de dimensiunea datelor de intrare şi de situaţia în care acesta este folosit.

References

[1] Tarjans off-line lowest common ancestors algorithm
`http://www.geeksforgeeks.org/tarjans-off-line-lowest-common-ancestors-algorithm/`

[2] Range Minimum Query and Lowest Common Ancestor
`https://www.topcoder.com/community/data-science/data-science-tutorials/range-minimum-query-and-low`

[3] Disjoint-set data structure
`https://en.wikipedia.org/wiki/Disjoint-set`$_{d}ata_{s}tructure$

[4] infoarena: Lowest Common Ancestor,
`http://www.infoarena.ro/problema/lca`