Analiza Algoritmilor

Man Andrei Vlad
Seria CD

Facultatea de Automatică şi Calculatoare
Universitatea Politehnica Bucureşti

1 Introducere

1.1 Descrierea problemei rezolvate

Cel mai mic strămoş comun a două noduri u şi v este nodul w, care este strămoş pentru ambele noduri şi are cea mai mare adancime T.

1.2 Aplicaţii practice

În practică, această problemă apare în ierarhiile moştenirilor speicifice Programării Orientată Obiect sau în sistemele complexe din programarea distribuită sau în grafică computaţională pentru afla cubul cel mai mic care conţine alte 2 cuburi.

1.3 Soluţiile alese

Soluţiile alese pentru aceasta problemă sunt algoritmii Binary Lifting şi Reducerea LCA la o problemă de tip RMQ.

Binary Lifting este o metodă prin care prepocesăm o matrice de dimensiuni [1,n]x[1,log(n)], unde elementul de pe pozitia (i,j) contine cel de-al 2 la j strămoş al nodului i. Prima oară verificăm dacă un nod este stramoşul celuilalt. În caz că este, cel mai mic stramoş comun devine chiar nodul respectiv. Altfel, ne mutăm mai sus, aducănd ambele noduri la acelaşi nivel, folosindu-ne de puterile lui 2. Apoi parcurgem arborele cu puterile maxime ale lui 2 pană ajungem sub cel mai mic strămoş comun. Complexitatea este O(NlogN + MlogN)

Reducerea LCA la o problemă de tip RMQ presupune traversarea euleriană a arborelui de la rădăcină, care este de fapt o traversare DFS cu caracteristici de traversare preordine. Nodul căutat este nodul de pe cel mai mic nivel dintre toate nodurile care apar între apariţii consecutive ale oricăror 2 noduri u si v din turul Euler. Complexitatea este O(N + MlogN).

1.4 Criteriile de evaluare pentru soluţia propusă

Se citesc numerele întregi N şi M, reprezentănd numărul de noduri ale arborelui T, respectiv numărul de interogări. În continuare se citesc N - 1 numere naturale, cel de-al i-lea număr reprezentănd tatăl nodului i + 1 (nodul 1 fiind rădăcină, nu are tată). Pe următoarele M linii se află căte o pereche de numere naturale, reprezentănd interogarea curentă.

Se vor afişa M numere naturale, al i-lea număr reprezentănd cel mai mic strămoş comun al nodurilor din interogarea i.

Testele se vor genera pe mai multe criterii in functie de N si M:

1. N mult mai mare decat M
2. N proporţional cu M
3. N mult mai mic decat M

Pentru N şi M valorile vor fi cuprinse intre 100 si 1.000.000, asiguranduse o varietate cat mai mare de teste.

Se vor evalua algoritmii in funcţie de timpul de execuţie şi de memoria consumată.

2 Prezentarea soluţiilor

2.1 Algoritmul de Binary Lifting

Pentru fiecare nod vom precalcula strămoşul lui, strămoşul de deasupra cu 2 noduri, cu 4 noduri (etc). Aceştia vor fi păstraţi într-o matrice memo, unde memo[i][j] este al 2 la j strămoş al nodului i, cu i de la 1 la N iar j de la 0 la ceil(log N). Astfel cu aceste informatii putem ajunge de la nod la un strămoş in timp O(logN). Această matrice o putem calcula printr-o parcurgere DFS.

Algorithm 1: DFS traversal

Data:
memo: matrix of ancestors
lev: node level array
log: heigth of tree (ceil(log2(n))
g: treeset
Function `DFS`(u, p)**:**
    memo[u][0] = p;
    for $i = 1;\ i < log;\ i++$ **do**
        memo[u][i] = memo[memo[u][i - 1]][i - 1];
    end
    for $v : g[u]$ **do**
        if $v! = p$ **then**
            lev[v] = lev[u] + 1;
            DFS(v, u);
        end
    end

Astfel, se calculează în jur de log N strămoşi pentru fiecare nod, fiind N noduri în total. Numărul operaţiilor per fiecare bucla este mic deci putem sa nu ţinem cont de acesta. Complexitatea temporală pentru precalculare este aproximativ O(NlogN), iar spaţiul ocupat este de asemenea O(NlogN), ocupat de matricea memo.

Se primeşte un query de două noduri u şi v. Verificăm dacă unul dintre noduri este strămoşul celuilalt. Dacă nu este, atunci găsim un nod care nu este strămoş comun pentru ambele noduri dar care este cel mai înalt nod. După ce găsim un astfel de nod, să zicem x, memo[x][0] va fi rezultatul căutării. Acest proces va necesita la randul său un timp O(logN).

Algorithm 2: LCA Query

```
Data:
memo: matrix of ancestors
lev: node level array
log: heigth of tree (ceil(log2(n))
g: treeset
Function LCA(u, v):
    if lev[u] < lev[v] then
        swap(u, v);
    end
    for i = log; i > 0; i − − do
        if lev[u] − 2^i >= lev[v] then
            u = memo[u][i];
        end
    end
    if u == v then
        return u
    end
    for i = log; i > 0; i − − do
        if memo[u][i]! = memo[v][i] then
            u = memo[u][i];
            v = memo[v][i];
        end
    end
    return memo[u][0]
```

Cu un timp de logN per query, complexitatea finală pentru acest proces va fi O(MlogN). Astfel o complexitate totală acestui algoritm va fi O(NlogN + MlogN), ocupând un spaţiu auxiliar de O(NlogN).

2.2 Reducerea LCA la o problemă de tip RMQ

Pentru fiecare nod vom calcula si reţine nivelul pe care se află într-un vector level. Se face parcurgerea euleriană si se creează un vector eulerian. Se generează inca 2 vectori l si h. Se va contrui un Segment Tree.

Algorithm 3: Euler traverse(DFS)

```
Data:
e: Euler array
g: Adjacency matrix
visited: Visited nodes matrix
Function DFS(root):
    e pushback root;
    visited[root] = True;
    for i = 0; i < g[root].size(); i++ do
        destination = g[root][i];
        if !visited then
            DFS(destination);
            e pushback root;
        end
    end
```

Algorithm 4: SegmentTree

```
Data:
e: Euler array
g: Adjacency matrix
visited: Visited nodes matrix
l: Level matrix
Function SegmentTree(start,end,i):
    if start > end then
        return;
    end
    if start == end then
        st[i] = start ;
        return;
    end
    left = 2 * i + 1 ;
    right = 2 * i + 2 ;
    mid = (start + end) rshift 1 ;
    SegmentTree(start, mid, left) ;
    SegmentTree(mid+1, end, right) ;
    if l[st[left]] < l[st[right]] then
        st[i] = st[left] ;
    else
        st[i] = st[right];
    end
```

Precalcularea algoritmului va fi O(N) atat temporal cat şi spaţial. Mai departe se va rezolva prin tehnica RMQ fiecare query de noduri.

Algorithm 5: RMQ

```
Data:
level: Level matrix
st: Segment tree matrix
Function RMQ(start,end,minNode,maxNode,i):
    if start > end then
        return -1;
    end
    if end < minNode OR maxNode < start then
        return -1;
    end
    if minNode <= start AND end <= maxNode then
        return st[i];
    end
    left = 2 * i + 1 ;
    right = 2 * i + 2 ;
    mid = (start + end) rshift 1 ;
    st = SegmentTree(start, mid, minNode, maxNode, left) ;
    en = SegmentTree(mid+1, end, minNode, maxNode, right) ;
    if st! = −1 AND en! = −1 then
        if level[st] < level[en] then
            return st;
        end
        return en;
    else if st! = −1 then
        return st;
    else if en! = −1 then
        return en;
    return 0;
```

Fiecare query are o complexitate O(logN). În final rezultă o complexitate temporala totala O(N + MlogN), N va fi numărul de noduri iar M numărul de query-uri, iar spaţial va folosi O(N).

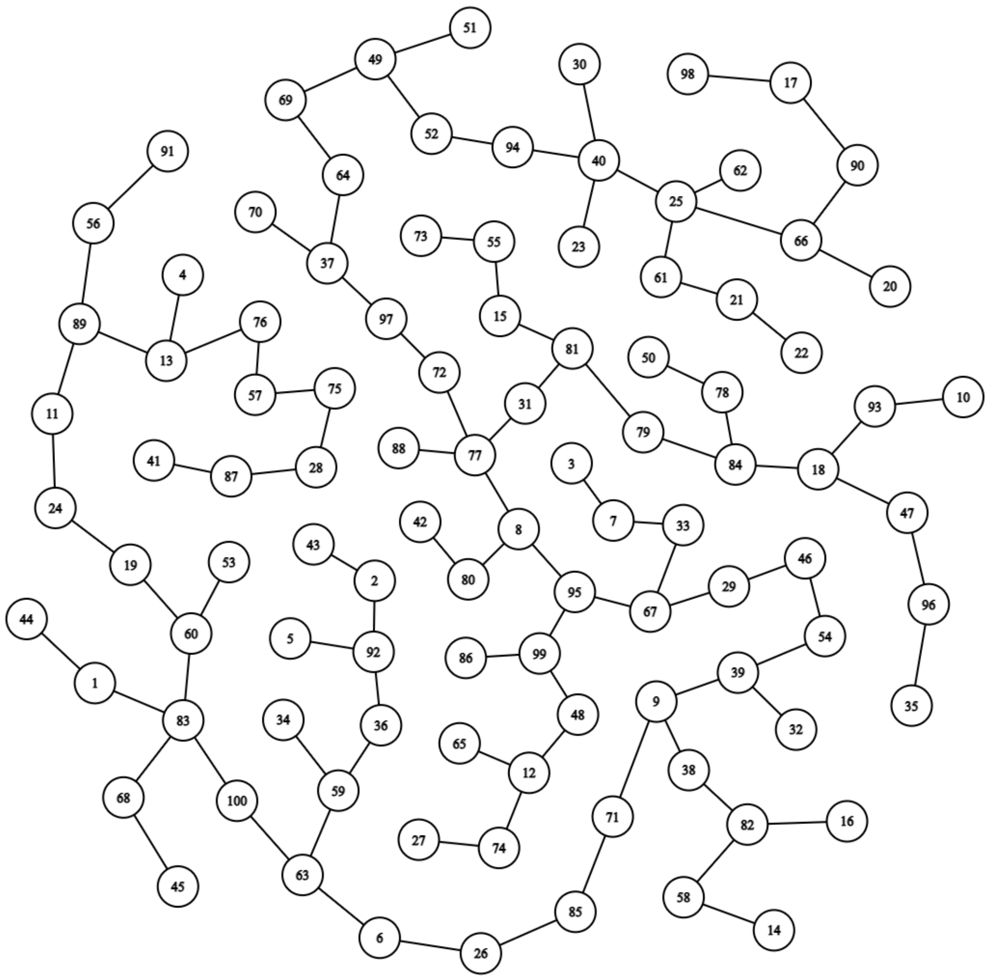
2.3 Avantaje si dezavantaje ale algoritmilor

Din punct de vedere al timpului de execuţie, algoritmii au rezultate similare. Astfel, departajarea se face prin spaţiul ocupat, Binary Lifting utilizand O(NlogN) din punct de vedere al resurselor iar RMQ doar O(N). În testele propuse, totuşi, primul algoritm are rezultate mai bune cu aproximativ 10-15% (vom observa in următoarea parte).

3 Evaluare

3.1 Modalitatea construirii setului de teste

Cele 20 de teste de intrare au fost generate folosind un algoritm ce asigură o generare aciclică, cu arbori cu pană la 1.000.000 de noduri, însa aceştia nu vor fi balansaţi. Apoi query-urile au fost generate random. Aşa arată reprezentarea grafului de la testul 1, avand 100 de noduri (cu varful in 1, stanga-jos).



Testele au valori pentru N între 100 şi 100.000, fiecare avand la randul său valori pt M între 100 si 1.000.000. Algoritmul folosit pentru generare este unul care utilizează Secvenţa Prufer.

3.2 Specificicaţiile sistemului de calcul

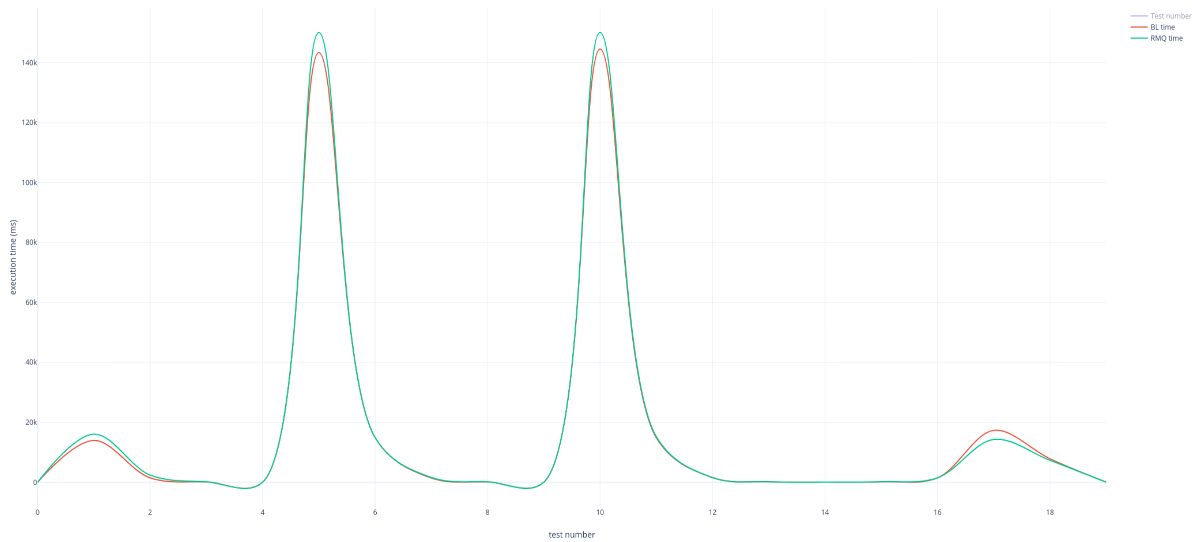
Hardware
- CPU: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz
- RAM: 16.0 GB @ 2400 MHz
- Storage: 952GB SSD

3.3 Teste propuse şi rezultate

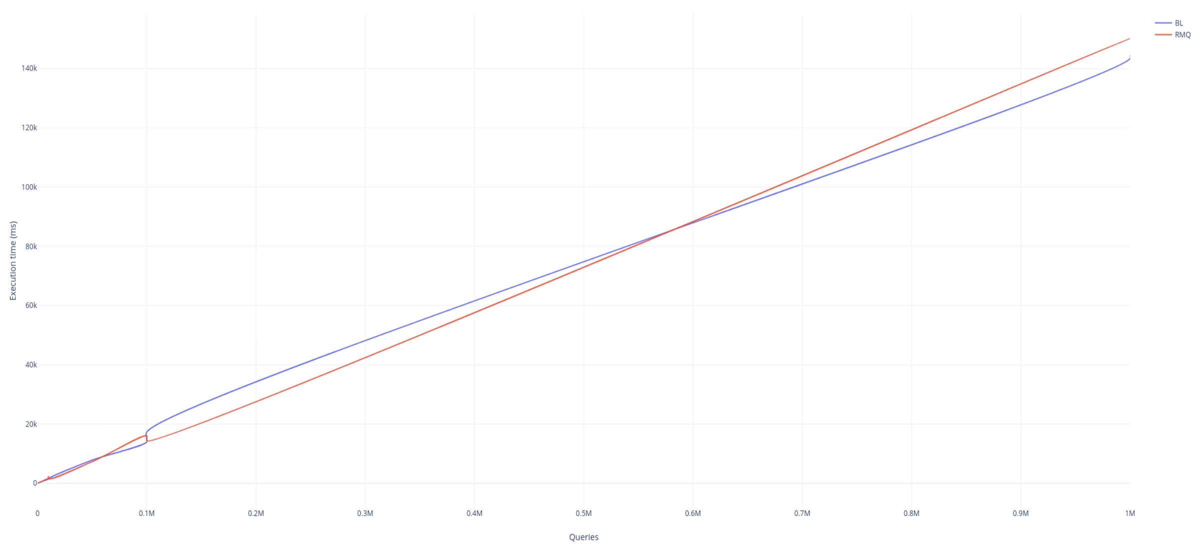
Următorul tabel conţine atat datele de intrare ale testelor căt şi timpii de execuţie. Observăm că, deşi în teorie, al doilea algoritm este mai puternic cu puţin, testele spun ca acesta este mai lent decat primul cu 13.7%.

Nr test	N (nodes)	M (queries)	Alg BL (ms)	Alg RMQ (ms)	Performance
1	100	100	21	24	14.3%
2	1000	100000	13940	16073	15.3%
3	1000	10000	1472	2402	63.2%
4	1000	1000	139	155	11.5%
5	1000	100	13	18	38.5%
6	10000	1000000	143371	150100	4.7%
7	10000	100000	14941	14943	0.0%
8	10000	10000	1409	1610	14.3%
9	10000	1000	149	180	20.8%
10	10000	100	21	23	9.5%
11	100000	1000000	144517	150084	3.9%
12	100000	100000	14854	15200	2.3%
13	100000	10000	1570	1541	-1.8%
14	100000	1000	189	165	-12.7%
15	100000	100	49	52	6.1%
16	100	1000	161	198	23.0%
17	100	10000	1454	1478	1.7%
18	100	100000	17320	14298	-17.4%
19	50000	50000	7834	7335	-6.4%
20	50000	50	24	44	83.3%
				In medie:	13.7%

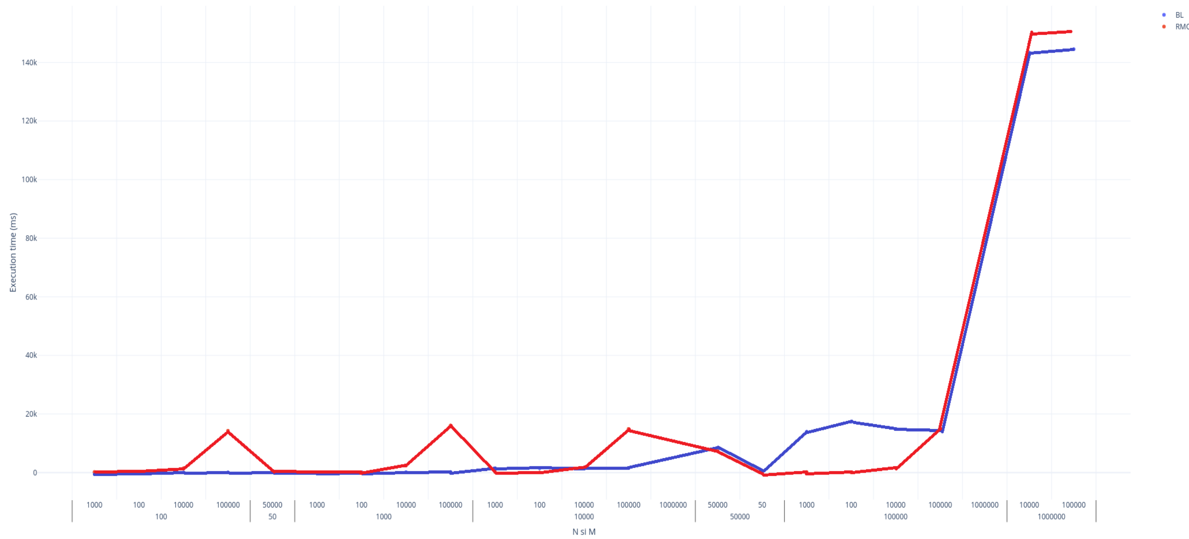
Graficul de mai jos ilustrează evoluţia timpului de executare in funcţie de numărul testului. În cele 2 vărfuri ale graficului sunt valorile maximale ale lui M, respectiv 1.000.000.



Următorul grafic ne arată o comparaţie între timpii de execuţie în funcţie de M, numărul query-urilor.



Ultimul grafic ne arată atat in funcţie de N cat şi de M timpul de executie, fiind reprezentativ pentru complexitate.



4 Concluzie

Aşadar, algoritmii aleşi, deşi au metode diferite de rezolvare, au o complexitate similară. Din punct de vedere al timpului de execuţie, aş alege să folosesc primul algoritmul, care teoretic e mai lent dar în practic e mai rapid cu 13% în cele mai multe cazuri. Din punct de vedere al spaţiului folosit, al doilea algoritm foloseţe mult mai puţine resurse deci este clar alegerea potrivită. Ultimul algoritm poate fi considerat superior primului datorită micilor diferenţe în timpul execuţiei din punct de vedere al timpului.

5 Bibliografie

https://www.geeksforgeeks.org/lca-in-a-tree-using-binary-lifting-technique/
https://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-tree-set-3-using-rmq/
https://www.geeksforgeeks.org/random-tree-generator-using-prufer-sequence-with-examples/