Cel mai mic strămoş comun

Constantin Mihai

Grupă: 321CD
mihai.constantin98@gmail.com
Facultatea de Automatică şi Calculatoare
Universitatea Politehnica Bucureşti

Rezumat Se doreşte analizarea problemei celui mai mic strămoş comun din teoria grafurilor. Se vor compara doi algoritmi de rezolvare a acesteia şi se vor calcula complexităţile în fiecare caz.

Keywords: Parcurgere Euler · RMQ · DFS · Preprocesare · Queries.

1 Introducere

1.1 Descrierea problemei

Se dă un **arbore T**. Cel mai apropiat strămoş comun a două noduri **u** şi **v** este nodul **w** care este strămoş al ambelor noduri **u** şi **v** şi are cea mai mare adâncime din **T**.

Considerăm că arborele **T** are **N** noduri şi are rădăcina în nodul 1. Dându-se o mulţime arbitrară P = {{u, v}}, cu **M** perechi neordonate de noduri din **T**, se cere să se determine cel mai apropiat strămoş al fiecărei perechi din P.

1.2 Aplicaţii practice

Problema celui mai mic strămoş comun a fost studiată intens în a doua jumătate a secolului XX, şi este interesantă nu numai din prisma algoritmilor complecşi de rezolvare, ci şi din perspectiva numeroaselor aplicaţii în **procesarea de stringuri** sau în **biologia computaţională**, unde LCA este utilizat împreună cu arbori de sufixe şi alte structuri arborescente similare.

Dov Harel şi **Robert Tarjan** au fost primii care au studiat în amănunt această problemă, arătând că după o preprocesare liniară a arborelui, putem răspunde în timp constant la fiecare query în parte.

Să considerăm un exemplu mai puţin abstract: **arborele vieţii**. Este un fapt bine cunoscut că speciile actuale ale planetei au evoluat pe parcursul timpului. Această evoluţie poate fi reprezentată ca un arbore, în care nodurile reprezintă speciile, iar fii unor noduri reprezintă direct specia evoluată. Deoarece speciile cu caracteristici similare sunt divizate în grupuri, prin determinarea celui mai mic strămoş în acest arbore, putem găsi părintele comun a două specii, verificând dacă caracteristicile similare sunt moştenite de la acel părinte.

1.3 Specificarea soluţiilor alese

În această lucrare voi prezenta următorii algoritmi pentru a determina cel mai mic strămoş comun a două noduri:

Programare dinamică Considerăm matricea $A[0 : \log_2 N][1 : N]$, unde $A[i][j]$ reprezintă al 2^i strămoş al nodului j. Pentru a forma matricea A, folosim următoarea relaţie de recurenţă:

$$A[i][j] = \begin{cases} T[j],\ i = 0 \\ A[i-1][A[i-1][j]],\ i > 0 \end{cases}$$

Am considerat vectorul $T[1 : N]$, unde T[i] reprezentă părintele nodului i.

Pentru fiecare query, se aduce nodul de pe nivelul mai mare pe acelaşi nivel cu celălalt în timp logaritmic, după care, tot în timp logaritmic, se poate afla LCA-ul celor două noduri.

Reprezentare Euler. RMQ O soluţie mai optimă decât cea prezentată anterior se foloseşte de reprezentarea Euler a unui arbore.
Considerăm următorii vectori:

- $E[1 : 2*N-1]$, unde $E[i]$ reprezintă al i-lea nod vizitat din **T**
- $L[1 : 2*N-1]$, unde $L[i]$ reprezintă nivelul nodului $E[i]$
- $H[1 : N]$, unde $H[i]$ reprezintă indexul primei apariţii a nodului i în E

Soluţia se bazează pe următoarea observaţie: cel mai apropiat strămoş comun a două noduri este nodul de nivel minim dintre primele apariţii ale nodurilor din query din reprezentarea Euler a arborelui.

Pentru a implementa această soluţie se pot folosi arbori de intervale, sau mai eficient, pentru determinarea minimului unei subsecvenţe se poate utiliza RMQ.

1.4 Criterii de evaluare

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

Date de iesire. **M** numere naturale, al i-lea număr reprezentând cel mai mic strămoş comun al nodurilor din interogarea i.

Seturile de date. Pentru a face o analiză cât mai obiectivă a eficienţei algoritmilor aleşi, vom discuta următoarele cazuri:

- $N << M$ (numărul de noduri este mult mai mic decât numărul de interogări)
  - $N = 200\ 000$
  - $M = 2\ 000\ 000$
- $N \simeq M$ (numărul de noduri este proporţional cu numărul de interogări)
  - $N = 200\ 000$
  - $M = 200\ 000$
- $N >> M$ (numărul de noduri este mult mai mare decât numărul de interogări)
  - $N = 200\ 000$
  - $M = 1\ 000$

Pentru fiecare set în parte vom analiza atât memoria necesară, cât şi timpul de execuţie.

2 Prezentarea soluţiilor

2.1 Descrierea modului în care funcţionează algoritmii aleşi

Programare dinamică Arborele T este reprezentat printr-o listă înlănţuită alocată dinamic de forma: $vector < int > graph[dmax]$, unde $dmax$ reprezintă dimensiunea maximă a numărului de noduri. După citirea numărului de noduri N şi a numărului de interogări M, se citesc cele $N-1$ muchii ale arborelui şi se adaugă în structură.
În continuare am parcurs lista vecinilor fiecărui nod pentru a forma vectorul $parent[dmax]$, cu $parent[i]$ reprezentând părintele nodului i.

Cheia algoritmului constă în crearea matricei $A[0 : \log_2 N][1 : N]$, având următoarea semnificaţie: $A[i][j]$ reprezintă al 2^i strămoş al nodului j. Ne vom folosi de următoarea recurenţă:

$$A[i][j] = \begin{cases} parent[j],\ i = 0 \\ A[i-1][A[i-1][j]],\ i > 0 \end{cases}$$

Prima relaţie este evidentă, deoarece primul strămoş al fiecătui nod este chiar părintele acestuia. Cea de-a doua relaţie poate fi interpretată astfel: al 2^i strămoş al nodului j reprezintă al 2^{i-1} strămoş al celui de-al 2^{i-1} strămoş al lui j.

După ce am format matricea A, am realizat o parcurgere DFS a arborelui pentru a obţine vectorul $level[dmax]$, unde $level[i]$ reprezintă înălţimea nodului i din arbore.

În continuare, pentru fiecare query, se aduce nodul de pe nivelul mai mare pe acelaşi nivel cu celălalt în timp logaritmic, după care, tot în timp logaritmic, se poate afla LCA-ul celor două noduri.

Reprezentare Euler. RMQ Arborele T este reprezentat printr-o listă înlănţuită alocată dinamic de forma: $std :: vector < std :: vector < int >> graph$. După citirea numărului de noduri N şi a numărului de interogări M, se citesc cele $N - 1$ muchii ale arborelui şi se adaugă în structură.

În prima etapă a algoritmului se realizează o parcurgere euleriană a arborelui. La terminarea apelului recursiv, avem formaţi următorii vectori:

- $E[1 : 2 * N - 1]$, unde $E[i]$ reprezintă al i-lea nod vizitat din **T**
- $H[1 : N]$, unde $H[i]$ reprezintă indexul primei apariţii a nodului i în E
- $level[1 : N]$, unde $level[i]$ reprezintă înălţimea nodului i din arbore

În acest moment, ne putem forma şi vectorul $L[1 : 2 * N - 1]$, unde $L[i]$ reprezintă nivelul nodului $E[i]$.

Ideea algoritmului constă în următoarea afirmaţie: cel mai apropiat strămoş comun a două noduri este nodul de nivel minim dintre primele apariţii ale nodurilor din query din reprezentarea euleriană a arborelui. Am redus problema la aflarea minimului între două poziţii ale unui şir (Range Minimum Query).
Vom folosi următoarele structuri:

- $lg[1 : 2 * N - 1]$, unde $lg[i]$ reprezintă $[\log_2 i]$
- $rmq[1 : 2 * N - 1][0 : \log_2(2 * N - 1)]$, unde $rmq[i][j]$ reprezintă indexul minimului dintr-un interval de lungime 2^j care se termină pe poziţia i

Pentru fiecare interogare de forma (p, q), determin indexul minimului dintr-un interval de lungime $q - p + 1$ care se termină pe poziţia q ($p < q$). Soluţia pentru $lca(p, q)$ va fi în $E[idx]$.

2.2 Analiza complexităţii soluţiilor

Programare dinamică Algoritmul de faţă reţine pentru fiecare nod strămoşul cu 2^i nivele mai sus, unde i ia valori între 1 şi $LEVEL$, unde $LEVEL$ este o constantă care ne spune valoarea maximă pentru al 2^i-lea strămoş posibil. Pentru a determina valoarea constantei $LEVEL$ vom analiza cazul cel mai defavorabil. Acesta apare în momentul în care fiecare nod din arbore are maxim un parinte, respectiv un copil. Cu alte cuvinte, arborele este degenerat într-o listă înlănţuită. Rezultă că $LEVEL = ceil(log(number\_of\_nodes))$.

De asemenea, facem o **precalculare** pentru a determina înalţimea fiecărui nod din arbore, cu ajutorul unei parcurgeri în adâncime. Aceasta se realizează într-un timp liniar $O(numberOfNodes) = O(N)$.

Algorithm 1 Preprocesare. Crearea matricei A

procedure MATRIX($N, parent[MAXN]$)
  for $i = 2$ to N **do**
    $A[0][i] = parent[i]$ ▷ primul strămoş al nodului i este $parent[i]$

  for $k = 1$ to $LEVEL$ **do**
    for $i = 1$ to N **do**
      $A[k][i] = A[k-1][A[k-1][i]]$ ▷ programare dinamică

  return 0

Complexitatea temporală a codului de mai sus este dată de cele două bucle for. Aceasta este $O(numberOfNodes * LEVEL) \sim O(N * \log_2 N)$.

Pentru a calculata LCA-ul a două noduri x şi y procedăm astfel:

- Primul pas este de a aduce nodurile la acelaşi nivel. Fie nodul x nodul cu adâncimea mai mare ($level[x] > level[y]$). Acum tot ceea ce trebuie să facem este să urcăm nodul x în arbore până la $level[y]$. Acest lucru se poate realiza în $O(\log_2(level[x] - level[y]))$, deoarece avem precalculat în matricea A al 2^i-lea strămoş pentru fiecare nod i.
- În acest moment x şi y sunt situate la acelaşi nivel. Din nou, ne vom folosi de strategia de a urca în arbore în timp logaritmic pe baza matricei A pentru a atinge primul strămoş comun al celor două noduri.

Algorithm 2 Cel mai mic strămoş comun

procedure LCA($N, parent[MAXN]$)
  for $k = LEVEL$ to 0 **do**
    if $A[k][x] \neq A[k][y]$ **then**
      $x = A[k][x]$
      $y = A[k][y]$
  return 0

Complexitatea temporală. Timpul per interogare este unul logaritmic. Deoarece avem M interogări, va rezulta o complexitate de $O(M * \log_2 N)$ pentru a răspunde la toate cele M query-uri.
Timpul necesar precalculării este $O(N * \log_2 N)$. Aşadar, complexitatea totală din punctul de vedere al timpului consumat este $O(N * \log_2 N + M * \log_2 N)$.

Complexitatea spaţială

- $O(N * \log_2 N)$, pentru memorarea strămoşilor fiecărui nod din arbore
- $O(N)$, pentru vectorii *level* şi *parent* care memorează înălţimile, respectiv părintele fiecărui nod din arbore

Complexitatea spaţială totală este $O(N * \log_2 N)$.

Reprezentare Euler. RMQ Vom considera nodul 1 rădăcină şi vom realiza o parcurgere euleriană a arborelui. În urma apelului recursiv, avem deja formaţi vectorii $E[1 : 2*N-1]$, $H[1 : N]$ şi $level[1 : N]$ cu semnificaţiile descrise mai sus.

Algorithm 3 Parcurgere Euler

```
procedure EULER(node, graph)
    visited[node] = true                  ▷ marchez drept vizitat nodul curent
    E[++K] = node                         ▷ adaug nodul în parcurgerea Euler
    if H[node] == 0 then
        H[node] = K                       ▷ reţin prima poziţia a nodului din parcurgerea Euler

    for i = 0 to graph[node].size() − 1 do
        y = graph[node][i]                ▷ y este vecin al lui node
        if visited[y] == false then
            level[y] = 1 + level[x]       ▷ calculez înalţimea din arbore a nodului y
            euler(y, graph)               ▷ apel recursiv
            E[++K] = node                 ▷ adaug nodul în parcurgerea Euler
    return 0
```

Complexitatea temporală a codului de mai sus este $O(numberOfNodes) = O(N)$, deoarece iterăm prin fiecare nod o singură dată (fapt asigurat de vectorul $visited$). Algoritmul este practic, o **parcurgere în adâncime** a arborelui.

De asemenea, în acest moment ne putem forma şi vectorul $L[1 : 2*N-1]$, unde $L[i]$ reprezintă înălţimea nodului $E[i]$.
Pentru a calculata LCA-ul a două noduri x şi y procedăm astfel:

- determinam poziţia minimului (idx) din intervalul L$[H[x] : H[y]]$ cu ajutorul algoritmului **Range Minimum Query**
- vom răspunde în $O(1)$ la fiecare query, deoarece cel mai mic strămoş comun al nodurilor x şi y va fi $E[idx]$

Complexitatea temporală

- Parcurgere Euler: Numărul de noduri este N. Pentru un arbore, numărul de muchii este $N-1$. Complexitatea pentru parcurgerea Euler va fi $O(2*N-1)$, adică $O(N)$.
- RMQ: Ne vom folosi de o matrice $rmq[1 : 2*N-1][0 : \log_2(2*N-1)]$, cu următoarea semnificaţie: $rmq[i][j]$ reprezintă indexul minimului dintr-un interval de lungime 2^j care se termină pe poziţia i. Complexitatea este $O(N * \log_2 N)$.
- Query: pentru fiecare query vom răspunde în $O(1)$, pe baza matricei rmq. Deoarece avem M query-uri, vom răspunde la toate în $O(M)$.

Complexitatea temporală totală este $O(N * \log_2 N + M)$.

Algorithm 4 Range Minimum Query

procedure RMQ($N, E, level$)
  lg[1] = 0; ▷ $\log_2 1 = 0$
  for $i = 2$ to $2 * N - 1$ **do**
    $lg[i] = lg[i/2] + 1$ ▷ $\log_2 i = \log_2 \frac{i}{2} * 2 = \log_2 \frac{i}{2} + \log_2 2 = \log_2 \frac{i}{2} + 1$

▷ poziţia minimului dintr-un interval de lung. 1 care se termină pe poziţia i este i
  for $i = 1$ to $2 * N - 1$ **do**
    $rmq[i][0] = i$
  for $i = 1$ to $2 * N - 1$ **do**
    for $j = 1$ to $[\log_2 i]$ **do**
      $rmq[i][j] = rmq[i][j-1]$
      if $L[rmq[i - 2^{j-1}][j-1]] < L[rmq[i][j]]$ **then**
        $rmq[i][j] = rmq[i - 2^{j-1}][j-1]$
  return 0

Complexitatea spaţială

- Pentru vectorul în care memorăm nodurile din parcurgerea euleriană, cât şi pentru vectorul L în care avem înălţimile corespunzătoare nodurilor din E avem o complexitate de $O(2 * N - 1) \sim O(N)$
- Pentru vectorul în care reţinem primele apariţii din E ale fiecărui nod, cât şi pentru vectorul *level* în care memorăm înalţimile nodurilor avem o complexitate egală cu $O(N)$
- Pentru matricea rmq avem o complexitate de $O(2 * N - 1 * \log_2(2 * N - 1))$, adică $O(N * \log_2 N)$
- Pentru vectorul lg în care memorăm valoarea lui $\log_2 i$, pentru i de la 1 la $2 * N - 1$ avem o complexitate de $O(N)$

Complexitatea spaţială totală este $O(N * \log_2 N)$.

2.3 Prezentarea principalelor avantaje şi dezavantaje pentru soluţiile alese

Programare dinamică Soluţia folosind matricea $A[0 : \log_2 N][1 : N]$, unde $A[i][j]$ reprezintă al 2^i strămoş al nodului j, are avantajul că este mult mai uşor de implementat în regim de concurs faţă de varianta care foloseşte parcurgere euleriană şi RMQ.
Faţa de soluţia optimă, acest algoritm răspunde la fiecare interogare în $O(\log_2 N)$, deoarece pentru fiecare query, se aduce nodul de pe nivelul mai mare pe acelaşi nivel cu celălalt în timp logaritmic, după care, tot în timp logaritmic, se poate afla LCA-ul celor două noduri. Precalcularea matricei A durează $O(N * \log_2 N)$, de unde o complexitate totală de $O(N * \log_2 N + M * \log_2 N)$ faţă de cea oferită de soluţia optimă de $O(N * \log_2 N + M)$.

Reprezentare Euler. RMQ Acest algoritm are avantajul că răspunde în $O(1)$ la interogare, spre deosebire de soluţia de mai sus care găseşte LCA-ul în timp logaritmic. Răspunsul la fiecare query este obţinut pe baza matricei $rmq[1 : 2*N-1][0 : \log_2(2*N-1)]$, unde $rmq[i][j]$ reprezintă indexul minimului dintr-un interval de lungime 2^j care se termină pe poziţia i.
De asemenea, complexitatea temporală este mai mică faţă de soluţia care foloseşte programarea dinamică: $O(N*\log_2 N+M)$ faţă de $O(N*\log_2 N+M*\log_2 N)$.
Pentru a determina poziţia minimului dintr-un interval m-am folosit de algoritmul Range Minimum Query. Un dezavantaj al acestei metode ar fi folosirea unui spaţiu suplimentar $(N*\log_2 N)$, pentru determinarea poziţiei minimului pe intervale de puteri ale lui 2, ceea ce poate fi un impediment în anumite cazuri, faţa de implementarea ce foloseţe arbori de intervale.
Memoria folosită de această soluţie este mai mare decât cea folosită de soluţia cu programare dinamică.

3 Evaluare

3.1 Construirea setului de date

Pentru verificarea corectitudinii algoritmilor implementaţi am folosit 15 teste (10 dintre ele sunt în folderul *in*, celelalte 5, în directorul *other_tests*).

Testele propuse sunt generate, respectând următoarele cazuri:

- $N << M$ (numărul de noduri este mult mai mic decât numărul de interogări)
  - test0.in: $N = 100\ 000, M = 2\ 000\ 000$
  - test1.in: $N = 100\ 000, M = 1\ 800\ 000$
  - test2.in: $N = 100\ 000, M = 1\ 400\ 000$
  - test3.in: $N = 100\ 000, M = 800\ 000$

- $N \simeq M$ (numărul de noduri este proporţional cu numărul de interogări)
  - test4.in: $N = 100\ 000, M = 100\ 000$
  - test5.in: $N = 50\ 000, M = 50\ 000$
  - test6.in: $N = 10\ 000, M = 10\ 000$

- $N >> M$ (numărul de noduri este mult mai mare decât numărul de interogări)
  - test7.in: $N = 60\ 000, M = 6\ 000$
  - test8.in: $N = 60\ 000, M = 1\ 000$
  - test9.in: $N = 30\ 000, M = 4\ 000$

Folderul *other_tests* conţine 5 teste cu următoarea structură:

– test0.in: $N = 1\ 000$, $M = 100$
– test1.in: $N = 10\ 000$, $M = 100$
– test2.in: $N = 40\ 000$, $M = 10\ 000$
– test3.in: $N = 50\ 000$, $M = 1\ 000\ 000$
– test4.in: $N = 50\ 000$, $M = 5\ 000$

3.2 Specificaţiile sistemului de calcul

Algoritmii au fost rulaţi pe un MacBook Pro 2017 cu un procesor de 3.1 GHz Intel Core i5 şi o memorie de 8GB.

3.3 Rezultatul evaluării soluţiilor pe setul de teste. Interpretare

Am rulat fiecare test de 1000 de ori, obţinând următorii timpi medii:

- Pentru cazul în care numărul de noduri N este mult mai mic decât numărul de interogări M, soluţia care foloseşte parcurgerea Euler este mult mai rapidă

Tabela 1. Mediile timpilor de execuie pe testele cu $N << M$

	Programare dinamică [ms]	Parcurgere Euler. RMQ [ms]
test0.in	1413.8	**114.99**
test1.in	1396.4	**109.72**
test2.in	1062.4	**103.36**
test3.in	962.82	**60.062**

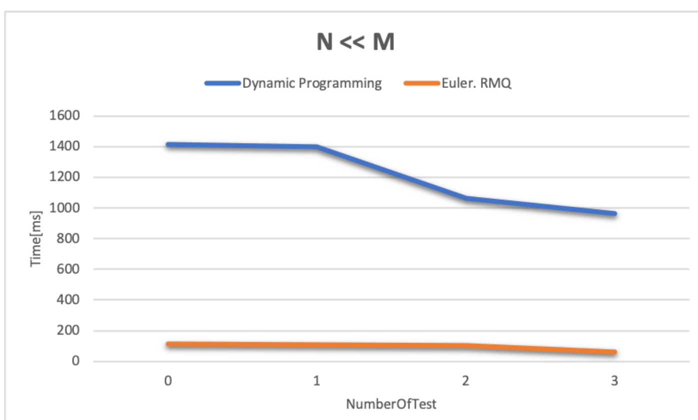

Fig. 1 Evoluţia timpilor pentru cazul $N << M$

- Pentru cazul în care numărul de noduri N este similar cu numărul de interogări M, tot algoritmul care foloseşte parcurgerea euleriană este mai rapid.

Tabela 2. Mediile timpilor de execuie pe testele cu $N \sim M$

	Programare dinamică [ms]	Parcurgere Euler. RMQ [ms]
test4.in	79.279	**37.112**
test5.in	52.908	**18.533**
test6.in	7.2306	**3.2453**

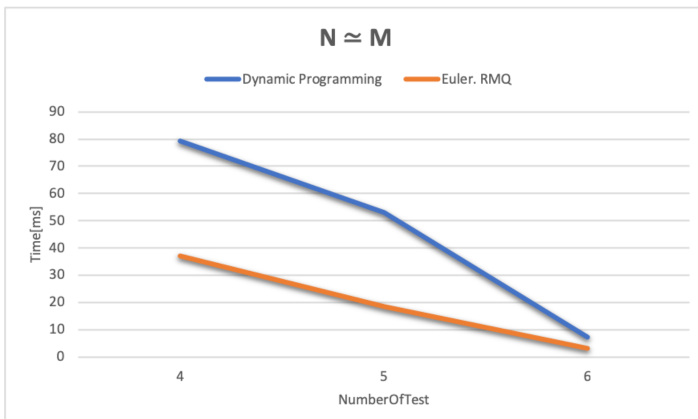

Fig. 2 Evoluţia timpilor pentru cazul $N \sim M$

- În cazul în care numărul de noduri N este mult mai mare decât numărul de interogări M, în mod surprinzător, algoritmul care foloseşte programarea dinamică se comportă mai bine pe setul de teste oferit.

Tabela 3. Mediile timpilor de execuie pe testele cu $N >> M$

	Programare dinamică [ms]	Parcurgere Euler. RMQ [ms]
test4.in	**12.011**	19.663
test5.in	**7.0011**	18.74
test6.in	**6.2517**	9.6168

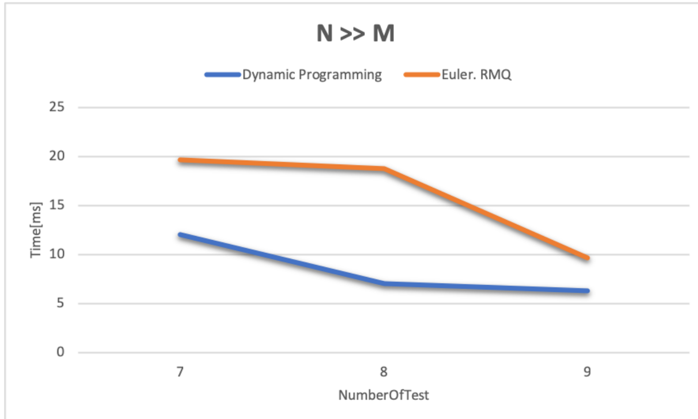

Fig. 3 Evoluţia timpilor pentru cazul $N >> M$

Timpii obţinuţi pe testele din folderul *other_tests* se pot observa mai jos. Remarcăm că soluţia ce foloseşte programarea dinamică se comportă şi aici mai bine pentru un număr mic de interogări.

Tabela 4. Mediile timpilor de execuţie pe testele din folderul *other_tests*

Numărul de noduri	Numărul de interogări	Programare dinamică [ms]	Parcurgere Euler. RMQ [ms]
1 000	100	**0.1539**	0.2456
10 000	100	**0.82587**	2.7526
40 000	10 000	**12.071**	13.074
50 000	5 000	**9.6711**	15.85
50 000	1 000 000	890.64	**57.069**

Reunind toate datele obţinute, atât pentru testele de bază, cât şi pentru cele din folderul *other_tests*, am obţinut următorul grafic:

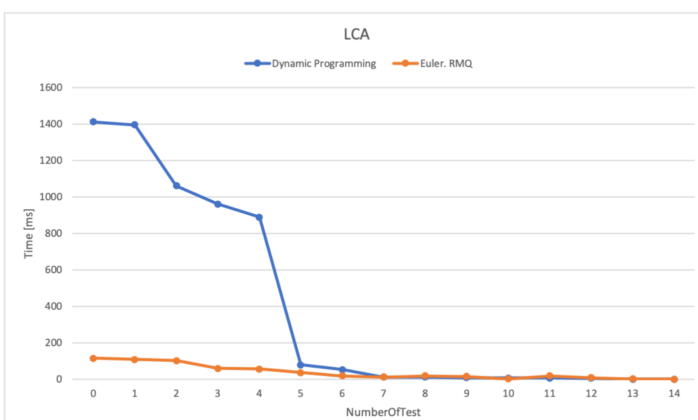

Fig. 4 Evoluţia timpilor de execuţie pentru testele propuse

Interpretare. Se observă că pentru cazurile în care numărul de noduri este mult mai mic sau similar cu numărul de interogări, soluţia folosind parcurgerea euleriană este mai bună. În momentul în care numărul de query-uri este mic, algoritmul ce foloseşte programarea dinamică se comportă foarte bine pe testele propuse, având performanţe mai bune din punct de vedere al timpului faţa de soluţia cu RMQ.

4 Concluzii

În concluzie, problema determinării celui mai mic strămoş comun poate fi rezolvată prin diverşi algoritmi a căror performaţă depinde de structura datelor de intrare. Aplicaţii practice ale acestei probleme sunt numeroase: procesare de string-uri, biologie computaţională etc.
În continuare prezint o aplicaţie pentru arborii cu costuri ce foloseşte problema celui mai mic strămoş comun într-un mod foarte elegant: Dându-se un arbore cu costuri să se răspundă rapid la întrebări de genul: *"Care este distanţa minimă între două noduri date?"*
O soluţie pentru problema de faţă ar fi:

- considerăm un nod oarecare rădăcină
- pentru fiecare nod calculăm distanţa până la rădăcină ($dist_i$)
- vom răspunde la fiecare interogare în $O(1)$, distanţa dintre oricare două noduri i şi j fiind egală cu $dist_i + dist_j - 2 * dist_{lca(i,j)}$

După cum s-a observat mai sus, soluţia ce foloseşte parcurgerea euleriană şi RMQ este mai rapidă în majoritatea cazurilor fâţă de algoritmul ce reţine pentru fiecare nod strămoşul cu 2^k nivele mai sus, unde k ia valori între 1 şi $\log_2 N$. În ciuda acestui lucru, pe testele oferite, în situaţia în care numărul de interogări este mult mai mic decât numărul de noduri, algoritmul ce foloseşte programare dinamică se comportă mai bine. De asemenea, acesta este mult mai uşor de implementat în regim de concurs fâţă de soluţia optimă. Personal, dacă numărul de interogări este foarte mare, de ordinul milioanelor, algoritmul cu RMQ este cel care trebuie implementat, iar dacă numarul de query-uri este mic, o abordare folosind metoda programării dinamice este de luat în calcul.

Bibliografie

1. Thomas H. Cormen, *Introduction to Algorithms*, Third Edition (2009)
2. G.L. McDowell, *Cracking the Coding Interview: 189 Programming Questions and Solutions*, 6th Edition, 2015
3. `https://infoarena.ro/lowest-common-ancestor` - data ultimei accesări: 14 decembrie 2018
4. `https://www.topcoder.com/community/competitive-programming/tutorials/range-minimum-query-and-lowest-common-ancestor` - data ultimei accesări: 14 decembrie 2018
5. `https://codeforces.com/blog/entry/53738` - data ultimei accesări: 14 decembrie 2018
6. `https://www.geeksforgeeks.org/find-lca-in-binary-tree-using-rmq` - data ultimei accesări: 14 decembrie 2018