Latvijas Universitāte

Datorikas fakultāte

Zemkvadrātisks algoritms koku ceļu apakšvirkņu uzdevumam

Maģistra darbs

Autors: Aleksejs Zajakins
Studenta apliecības nr.: az11182

Darba vadītājs: LU docents, Dr. dat. Jevgēnijs Vihrovs

Rīga 2021

Anotācija

Darbā tiek apskatīts garākās kopīgās apakšvirknes uzdevuma vispārinājums uz kokiem. Tiek apskatīti divi vienāda izmēra koki, kuru virsotnes ir marķētas ar vienas kopas elementiem. Garākā kopīgā apakšvirkne tiek meklēta pāri visiem ieejas koku ceļu pāriem. Galvenais rezultāts ir algoritms, kas risina apskatīto uzdevumu laikā $O(n \log^3 n)$, kur n ir abu ieejas koku virsotņu skaits.

Atslēgvārdi: grafi, koki, garākā kopīgā apakšvirkne, garākā augoša apakšvirkne, dinamiskā programmēšana, centroīdu dekompozīcija, Eilera apstaiga.

Abstract

Sub-Quadratic Algorithm for the Tree Path Subsequence Problem

In this work we consider a generalization of the longest common subsequence problem to trees. We examine two equally sized trees with the vertex labels in a common set. Specifically, we solve the problem of finding the longest common subsequence among all pairs of the input tree paths. The main result is an algorithm which solves the considered problem in $O(n \log^3 n)$ time, where n is the number of vertices in both input trees.

Keywords: graphs, trees, longest common subsequence, longest increasing subsequence, dynamic programming, centroid decomposition, Euler Tour Technique.

Autoreferāts

Visi 3. nodaļā aprakstītie paņēmieni nav autora oriģināli rezultāti, bet ir tikai vispārzināmu rezultātu atstāstījumi. 4.2. nodaļā aprakstītā datu struktūra ir paša autora izgudrojums, bet tās vispārīguma dēļ, tā visticamāk jau tika aprakstīta iepriekš un izmantota kādā citā kontekstā. Darba autors gan nav atradis nevienu citu rakstu, kurā būtu izmantota līdzīga datu struktūra, tāpēc 4.2. nodaļa nesatur atsauces uz avotiem. 4.3. un 5.1. nodaļās aprakstītie algoritmi ir galvenie darba rezultāti un tie ir autora oriģināli rezultāti.

Darba novitāte. Darba tika apskatīts garākās kopīgās apakšvirknes uzdevuma vispārinājums uz kokiem. Tika aprakstīts jauns algoritms, kas atrisina apskatīto uzdevumu laikā $O(n \log^3 n)$. Algoritma izstādes laikā viens no tā apakšuzdevumiem tika izdalīts kā atsevišķs vienkāršots uzdevums. Šim vienkāršotajam uzdevumam tika aprakstīts jauns algoritms, kas to risina laikā $O(n \log n)$. Šie divi jaunie algoritmi ir aprakstīti attiecīgi 5.1. un 4.3. nodaļās.

Literatūras izpēte. Darba izstrādes laikā tika izpētita saistīta literatūra, bet tā netika aprakstīta atsevišķā nodaļā. Atsauces uz izpētītajiem avotiem ir atrodamas darba ievadā.

Paveiktā praktiskā darba apjoms. Darba pamatā nav autora paša ieguldīts projektēšanas, programmēšanas vai cits praktisks darbs.

Rezultātu aprobācija. Parasti algoritmi, kas risina garākās kopīgās apakšvirknes uzdevumu (un tā atvasinājumus), ir balstīti uz dinamiskās programmēšanas pieejas. Darbā aprakstītie algoritmi arī ir balstīti uz dinamiskās programmēšanas pieejas, tāpēc tos var uzskatīt par līdzīgiem citiem rezultātiem šajā jomā. Darbā ir pierādīts, ka aprakstītie algoritmi atrisina uzstādītos uzdevumus zemkvadrātiskā laikā, kas arī bija darba mērķis. Darba rezultāti netika publicēti vai prezentēti konferencēs vai semināros. Darba rezultāti netiek izmantoti praksē. Darbā netika izstrādāta programmatūra, tāpēc tā netika testēta.

Darba noformējuma kvalitāte. Darba teksts ir vairākas reizes pārlasīts un pareizrakstība ir pārbaudīta ar attiecīgu programmatūru. Visas atrastās rakstības kļūdas ir izlabotas. Kur ir iespējams, darbā tika izmantota latviešu valodā oficiāli pieņemtā nozares terminoloģija. Lielākajai daļai izmantoto paņēmienu latviešu valodā nav oficiāli pieņemtu terminu, šādos gadījumos tika izmantota ikdienā lietotā terminoloģija (ja tāda eksistēja). Darba autors ir apskatījis norādījumu 6. pielikumu un ir izpildījis visus tā punktus, kas uzlabo darba noformējuma kvalitāti.

Plaģiāta iespējamība. Visi no citiem autoriem aizgūtie rezultāti ir atzīmēti ar attiecīgām literatūras atsaucēm. Daļai no 3. nodaļā aprakstītajiem rezultātiem nav atsauču, jo tiem nav konkrētu avotu un tie tiek uzskatīti par folkloru. Šādi 3. nodaļas rezultāti arī netiek pasniegti kā autora oriģināli rezultāti. Darbā nav teksta gabalu, kas ir burtisks tulkojums vai tuvs pārstāsts no kāda viena literatūras avota, bet daļa no definīcijām var precīzi sakrist ar kādiem citiem darbiem acīmredzamu iemeslu dēļ.

Saturs

Ievads

Garākās kopīgās apakšvirknes uzdevums (angliski *Longest Common Subsequence*, jeb LCS) ir viens no klasiskajiem datorzinātnē pētītajiem uzdevumiem. Tam ir vairāki praktiski pielietojumi dažādās jomās, ieskaitot datu salīdzināšanu, bioinformātiku, lingvistiku un ķīmiju. Vispārīgā gadījumā uzdevumam jau 1975. gadā ir aprakstīts dinamiskās programmēšanas algoritms ar lineāru telpas sarežģītību un kvadrātisku laika sarežģītību [1]. Gadu vēlāk tika pierādīts arī laika sarežģītības apakšējais novērtējums $\Omega(n^2)$ uzdevuma vispārīgajam gadījumam lēmumu koku (angliski *Decision Tree*) modelī [2]. Tajā pašā laikā, garākās kopīgās apakšvirknes uzdevums ir risināms zemkvadrātiskā laikā vairākos speciālgadījumos. Piemēram, gadījumā, ja abas ieejas virknes ir permutācijas, garākās kopīgās apakšvirknes uzdevums ir ekvivalents ar garākās augošas apakšvirknes uzdevumu (angliski *Longest Increasing Subsequence*, jeb LIS), kas jau ir risināms laikā $O(n \log n)$ [3].

Garākās kopīgās apakšvirknes uzdevumu var dabiski vispārināt arī uz citām struktūrām. Konkrēti, šajā darbā tiek apskatīts vispārinājums uz kokiem. Eksistē vairāki dažādi veidi, kā LCS uzdevumu var vispārināt uz kokiem, bet ir skaidrs, ka lielākajā daļā gadījumu iegūtais vispārinājums būs stingri sarežģītāks nekā oriģināls LCS uzdevums, un tātad vispārīgā gadījumā nebūs risināms zemkvadrātiskā laikā. Literatūrā jau tika apskatīti vairāki šādi vispārinājumi [4, 5, 6, 7, 8], kuros tiek meklēts lielākais kopīgais apakškoks, ko var iegūt no abiem ieejas kokiem izmantojot kaut kādas operācijas. Visiem aprakstītajiem algoritmiem sliktākā gadījuma laika sarežģītība ir $\Omega(n^2)$.

Darba mērķis ir atrast kādu dabisku LCS uzdevuma vispārinājumu uz kokiem, kas būtu risināms zemkvadrātiskā laikā. Lai šāds laiks būtu teorētiski sasniedzams, no paša sākuma tiek apskatīts tikai gadījums, kad katrā no ieejas kokiem nav divu vienādi marķētu virsotņu. Šis ierobežojums garantē, ka gadījumā, kad abi ieejas koki ir ceļi, tiek iegūts LCS uzdevuma gadījums ar permutācijām, nevis tā vispārīgais gadījums. Izvēlētais vispārinājums ir kaut kādā ziņā pats vienkāršākais — diviem patvaļīgiem kokiem tiek meklēta to garākā kopīgā ceļu apakšvirkne. Darba galvenais rezultāts ir jauns algoritms, kas risina izvēlēto uzdevumu laikā $O(n \log^3 n)$.

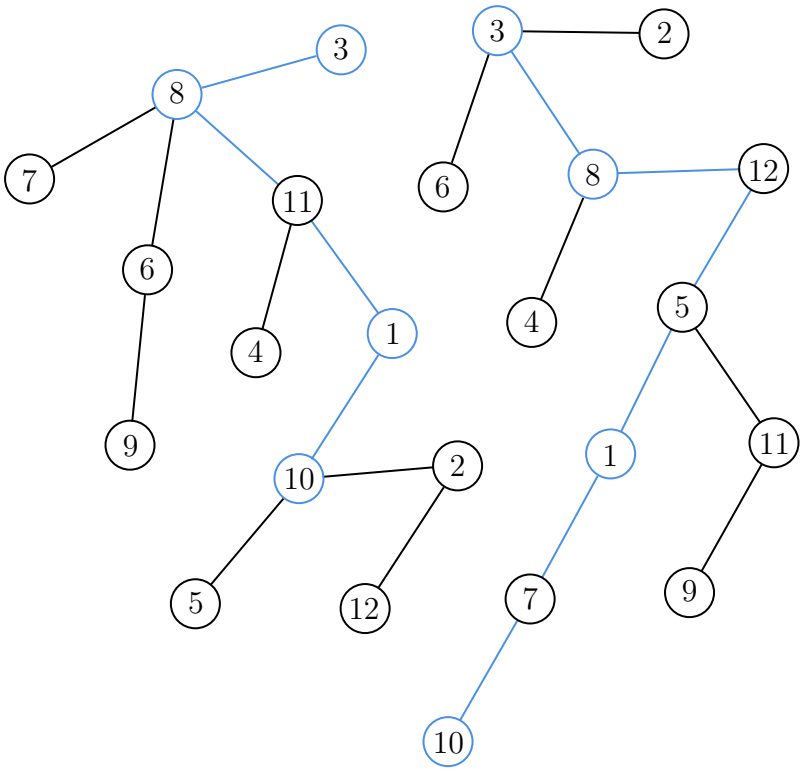
Darba saturīgā daļa ir sadalīta trīs daļās. 3. nodaļā ir aprakstīti paņēmieni, uz kuriem tiks balstīts risinājuma algoritms. 4. nodaļā ir aprakstīts vienkāršots uzdevuma variants un tā risinājums. Beidzot, 5. nodaļā ir aprakstīts pilna uzdevuma risinājums, kas iekšēji izmanto iepriekšējās nodaļas algoritmu.

1. Uzdevuma formulējums

Apskatīsim koku T un apzīmēsim ar $d(u, v)$ šķautņu skaitu uz vienīgā vienkārša ceļa starp virsotnēm u un v. Virsotņu virkni $v_1, v_2, \ldots, v_k$ sauksim par koka T *apakšvirkni*, ja $v_i \neq v_{i+1}$ visiem $1 \leq i < k$ un $d(v_1, v_k) = \sum_{i=1}^{k-1} d(v_i, v_{i+1})$. Citiem vārdiem, virsotnes $v_1, v_2, \ldots, v_k$ atrodas uz viena vienkārša ceļa kokā T, tieši tādā secībā.

Dotiem diviem kokiem T_1 un T_2 ar kopīgu virsotņu kopu ir jāatrod garākās kopīgās apakšvirknes garumu.

1.1. Piemērs



1. zīm.: Divu koku garākās kopīgās apakšvirknes piemērs

1. zīmējumā ir attēloti divi koki ar 12 virsotnēm un viena no to garākajām kopīgajām apakšvirknēm — $3, 8, 1, 10$. Garākā kopīgā apakšvirkne ne obligāti ir unikālā, piemērā virknes $10, 1, 8, 3$ un $10, 1, 11, 9$ arī ir abu attēlotu koku apakšvirknes ar garumu 4.

Piemērā sekojošas virknes ir abu koku apakšvirknes: $7, 8, 4$ un $9, 11, 12$. Virkne $5, 2, 12$ ir tikai pirmā koka apakšvirkne, savukārt virkne $5, 12, 2$ ir tikai otrā koka apakšvirkne.

2. Definīcijas

Strādājot ar kokiem, tiks izmantoti sekojoši apzīmējumi.

Definīcija 2.1. Virsotne u ir virsotnes v *kaimiņš* kokā T, ja koks T satur šķautni (u, v). Virsotnes v visu kaimiņu kopa tiks apzīmēta ar $\texttt{neighbors}(v)$.

Definīcija 2.2. Apskatīsim sakņotu koku T ar sakni r un tā patvaļīgu virsotni $v \neq r$. Par virsotnes v *vecāku* tiks saukts tās kaimiņš p, kas ir tuvāks saknei nekā v. Virsotnes v vecāks tiks apzīmēts ar $\texttt{parent}(v)$. Koka saknei $\texttt{parent}(r)$ nav definēts.

Definīcija 2.3. Ja sakņota koka virsotne v ir virsotnes u vecāks, tad u tiks saukta par virsotnes v *bērnu*. Virsotnes v visu bērnu kopa tiks apzīmēta ar $\texttt{children}(v)$.

Definīcija 2.4. Apskatīsim sakņotu koku T un sakni r un tā patvaļīgu virsotni v. Izdzēšot no koka T šķautni starp virsotnēm v un $\texttt{parent}(v)$, koks sadalās divās komponentēs. Tā komponente, kas satur virsotni v, tiks saukta par virsotnes v *apakškoku* un tiks apzīmēta ar $\texttt{subtree}(v)$. Koka saknei r, tā apakškoks ir definēts kā viss koks T, jeb $\texttt{subtree}(r) = T$.

Definīcija 2.5. Sakņota koka virsotnei v par tās *dziļumu* tiks saukts attālums (šķautņu skaits) starp virsotni v un koka sakni.

Definīcija 2.6. Par sakņota koka *dziļumu* tiks saukts maksimālais no koka virsotņu dziļumiem.

Definīcija 2.7. Koka T virsotņu kopa tiks apzīmēta ar $V(T)$.

Definīcija 2.8. Virsotnes v *pakāpe* ir tās kaimiņu skaits, jeb kopas $\texttt{neighbors}(v)$ izmērs. Virsotnes v pakāpe tiks apzīmēta ar $\deg v$.

Definīcija 2.9. Koka virsotne v ir *lapa*, ja tās pakāpe ir viens, jeb $\deg v = 1$.

Definīcija 2.10. Sakņota koka virsotnēm u un v par to *zemāko kopīgo priekšteci* (angliski Lowest Common Ancestor, jeb LCA) tiks saukta virsotne w ar maksimālu dziļumu, kurai ir spēkā, ka abas virsotnes u un v pieder virsotnes w apakškokam, jeb $\{u, v\} \subseteq V(\texttt{subtree}(w))$.

Aprakstot algoritmus, kas reizē apstrādā divus kokus, apzīmējumiem tiks lietoti apakšējie indeksi, lai paskaidrotu, uz kuru no ieejas kokiem tas apzīmējums attiecās. Piemēram, ja tiks apstrādāti koki T_1 un T_2, tad virsotnes v bērnu kopai kokā T_1 tiks izmantots apzīmējums $\texttt{children}_1(v)$. Arī datu struktūrām, kas attiecās tikai uz vienu no diviem ieejas kokiem, tiks lietoti apakšējie indeksi līdzīgā veidā.

Definīcija 2.11. Ar $[n]$ tiks apzīmēta naturālu skaitļu no 1 līdz n kopa, jeb $[n] = \{1, 2, \ldots, n\}$.

3. Palīgrīki un teorēmas

Šajā nodaļā aprakstīsim rezultātus, uz kuriem tiks balstīts algoritms.

3.1. Eilera apstaiga

Eilera apstaiga ir paņēmiens, kas ļauj vairākus uzdevumus uz kokiem reducēt uz ekvivalentiem vai līdzīgiem uzdevumiem uz lineāra masīva. Tā galvenā īpašība ir, ka sakņota koka virsotnes tiek sanumurētas ar skaitļiem no 1 līdz n tādā veidā, ka katras virsotnes v apakškokam tā virsotnēm ir piešķirti visi numuri kādā intervālā $[l, r]$. Šādā veidā apgalvojumus par oriģinālā koka apakškokiem var noformulēt izmantojot virsotņu numuru intervālus. Šis paņēmiens pirmo reizi tika aprakstīts [9] un angliski tika nosaukts par "*Euler Tour Technique* (ETT)". Šajā nodaļā paņēmiens netiks aprakstīts vispārīgā formā, bet tiks definēts jau pielāgoti konkrētiem pielietojumiem.

Definīcija 3.1. Apskatīsim koku T ar n virsotnēm un sakni r. *Eilera apstaiga* tiek definēta izmantojot divus masīvus – `EulerIn` un `EulerOut`, katrs izmērā n. `EulerIn` ir kopas $[n]$ permutācija. `EulerOut` elementi $\in [n]$. Masīvi apmierina sekojošu nosacījumu:

$$\texttt{EulerIn}[v] \leq \texttt{EulerIn}[u] \leq \texttt{EulerOut}[v]$$

tad un tikai tad, kad virsotne u atrodas virsotnes v apakškokā.

Piezīme. *Lielākajai daļai koku, pēc definīcijas 3.1, masīvi* ***`EulerIn`*** *un* ***`EulerOut`*** *nav vienozīmigi.*

Masīvus `EulerIn` un `EulerOut` var aizpildīt laikā $O(n)$, inicializējot $\texttt{counter} \leftarrow 0$ un izsaucot EulerTourDFS(r).

Algoritms 1 Eilera apstaiga

```
1: function EulerTourDFS(v)
2:     counter ← counter + 1
3:     EulerIn[v] ← counter
4:     for u ∈ children(v) do
5:         EulerTourDFS(u)
6:     end for
7:     EulerOut[v] ← counter
8: end function
```

3.2. Centroīdu dekompozīcija

Centroīdu dekompozīcija ir paņēmiens, kas ļauj patvaļīgu koku ar n virsotnēm sadalīt $O(n)$ sakņotos apakškokos ar kopējo izmēru $O(n \log n)$, pie tam iegūtajai dekompozīcijai ir vairākas noderīgas īpašības. Šī darba ietvaros būs aktuāla tikai viena īpašība – katrām divām oriģinālā koka virsotnēm u un v eksistē tieši viens dekompozīcijas apakškoks S ar sakni $r(S)$, ka $\{u, v\} \subseteq V(S)$ un virsotne $r(S)$ atrodas uz vienīgā vienkāršā ceļa starp u un v. Šāda veida koka dekompozīcija tiek izmantota, piemēram, rakstos [10, 11, 12]. Angliski to sauc par "*Centroid Decomposition*".

Teorēma 3.1 (Koka centroīds [13])**.** *Kokā ar n virsotnēm eksistē virsotne, izdzēšot kuru koks sadalās vairākās komponentēs, kur katras komponentes izmērs nepārsniedz $\frac{n}{2}$ virsotnes.*

Pierādījums. Apskatīsim sekojošu procesu. Sākotnēji izvēlas patvaļīgu virsotni v. Ja virsotne v apmierina teorēmas nosacījumu, tad teorēma ir pierādīta. Citādi, izdzēšot virsotni v, izveidojas tieši viena komponente ar vairāk nekā $\frac{n}{2}$ virsotnēm. Apskatīsim virsotni u, kas pieder šai komponentei, un ir savienota ar šķautni ar v. Virsotni v aizstāj ar virsotni u un atgriežas uz procesa sākumu. Procesa laikā nav iespējams atgriezties iepriekš apmeklētajā virsotnē, jo iepriekšējās virsotnes komponentes izmērs ir stingri mazāks par $\frac{n}{2}$. Tātad šis process nevar turpināties bezgalīgi, jo koka virsotņu skaits ir galīgs, un izbeigsies kādā virsotnē v, kas apmierina teorēmas nosacījumu. □

Definīcija 3.2 (Centroīdu dekompozīcija)**.** Apskatīsim patvaļīgu koku T, apzīmēsim tā centroīdu ar v. Ja, izdzēšot virsotni v no koka T, tiek iegūtas k komponentes $C_1, C_2, \ldots, C_k$, tad par koka T *centroīdu dekompozīciju* sauc sakņotu koku $\texttt{CD}(T)$, kura sakne ir v, un tās bērnu apakškoki tiek konstruēti rekursīvi kā koku $C_1, C_2, \ldots, C_k$ centroīdu dekompozīcijas, jeb $\texttt{CD}(C_i)$. Tātad iekš centroīdu dekompozīcijas $\texttt{CD}(T)$ tā sakne v ir savienota ar komponenšu C_i centroīdiem. Teiksim, ka koka $\texttt{CD}(T)$ saknei v atbilst viss oriģinālais koks T, līdzīgi koka $\texttt{CD}(C_i)$ saknei atbilst oriģinālais koks C_i utt. Apzīmēsim koku, kas atbilst $\texttt{CD}(T)$ virsotnei u ar $\texttt{CC}(u)$.

Teorēma 3.2. *Apzīmējot ar $V(T)$ koka T virsotņu kopu un $n = |V(T)|$, izpildās sekojošā nevienādība:*

$$\sum_{v \in V(\texttt{CD}(T))} |V(\texttt{CC}(v))| \in O(n \log n)$$

Pierādījums. Apzīmēsim virsotnes v bērnu kopu ar $\texttt{children}(v)$. No centroīda definīcijas seko, ka, ja $v \in V(\texttt{CD}(T))$ un $u \in \texttt{children}(v)$, tad $2 \cdot |V(\texttt{CC}(u))| \leq |V(\texttt{CC}(v))|$, tātad koka $\texttt{CD}(T)$ dziļums ir $O(\log n)$. No centroīdu dekompozīcijas definīcijas seko, ka $\sum_{u \in \texttt{children}(v)} |V(\texttt{CC}(u))| = |V(\texttt{CC}(v))| - 1$, tātad katrā koka $\texttt{CD}(T)$ dziļuma līmenī ir spēkā $\sum_u |V(\texttt{CC}(u))| \leq n$, kur summa ir pāri visām virsotnēm u šajā dziļuma līmenī. Sasummējot šādas nevienādības pāri visiem dziļumu līmeņiem iegūstam prasīto. □

3.3. Virtuālais koks

Definīcija 3.3. Dotam kokam T un tā virsotņu apakškopai $U \subseteq V(T)$, apskatīsim tādus kokus S, ka $U \subseteq V(S) \subseteq V(T)$ un koku S var iegūt no koka T, izmantojot operācijas

1. izdzēst lapu;
2. izdzēst virsotni ar pakāpi 2 un pievienot šķautni starp tās kaimiņiem.

No visiem šādiem kokiem S par koka T *virsotņu apakškopas* U *virtuālo koku* sauksim koku S ar mazāko virsotņu skaitu.

Teorēma 3.3. *Virsotņu apakškopas U virtuālo koku var iegūt, sākot ar koku T un patvaļīgi pielietojot izdzēšanas operācijas virsotnēm, kuras var izdzēst.*

Pierādījums. Apzīmēsim ar A virsotņu kopu, ko var izdzēst pirms virsotnes $v \in A$ izdzēšanas operācijas, un ar B – kopu pēc operācijas. Parādīsim, ka $(A \setminus \{v\}) \subseteq B$. $u \in A$ tad un tikai tad, kad $u \notin U$ un $\deg u \leq 2$. Veicot operāciju, kopa U nemainās un $\deg u$ nevar palielināties, tāpēc $u \in (A \setminus \{v\}) \implies u \in B$. □

Teorēma 3.4. *Apzīmējot $n = |V(T)|$ un $k = |U|$, koka T virsotņu apakškopas U virtuālo koku var iegūt laikā $O(k \log k)$ ar priekšapstrādes laiku $O(n)$.*

Pierādījums. Aprakstīsim algoritmu, kas atrisina šo uzdevumu prasītajā laikā. Priekšapstrādes solī tiek sagatavoti Eilera apstaigas masīvi `EulerIn` un `EulerOut` un datu struktūra zemākā kopīgā priekšteča (LCA) noteikšanai [14, 15]. Kā koka sakne tiek izvēlēta patvaļīga virsotne $r \in V(T)$. Dotai virsotņu apakškopai U, tās virtuālais koks tiek iegūts simulējot koka T Eilera apstaigu, bet apskatot tikai brīžus, kad tiek apmeklētas kopas U virsotnes. Precīzs algoritms ir definēts zemāk. □

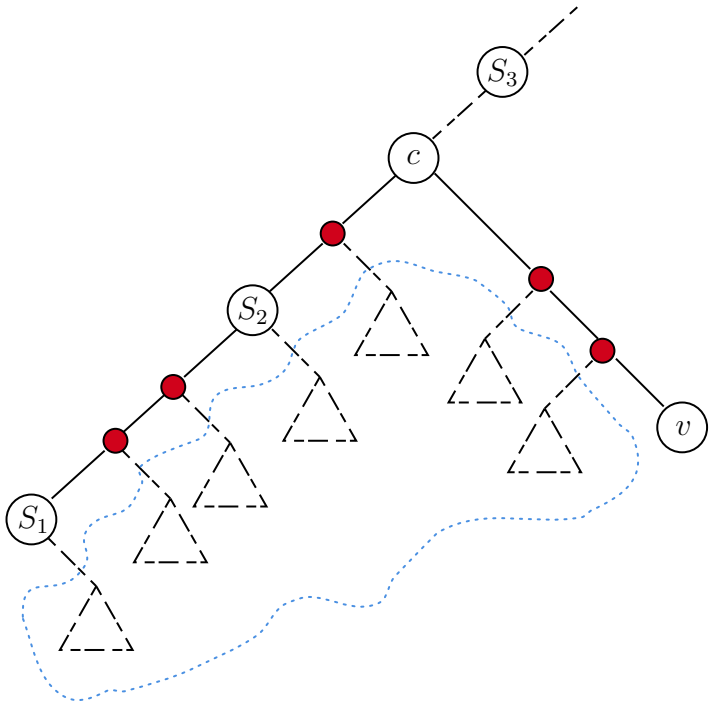
Algoritms 2 Virtuālā koka konstruēšana

```
1:  stack ← empty stack
2:  for v ∈ U in increasing order of EulerIn[v] do
3:      while stack.size ≥ 2 do
4:          a ← stack.Pop()
5:          b ← stack.Top()
6:          if EulerOut[b] < EulerIn[v] then
7:              Add edge (a, b) to the tree S
8:          else
9:              stack.Push(a)
10:             break
11:         end if
12:     end while
13:
```

```
    if stack.size > 0 then
        a ← stack.Pop()
        if EulerOut[a] < EulerIn[v] then
            c ← LCA(a, v)
            if stack.Top() ≠ c then
                Add vertex c to the tree S
                stack.Push(c)
            end if
            Add edge (a, c) to the tree S
        else
            stack.Push(a)
        end if
    end if

    Add vertex v to the tree S
    stack.Push(v)
end for

while stack.size ≥ 2 do
    a ← stack.Pop()
    b ← stack.Top()
    Add edge (a, b) to the tree S
end while

r ← stack.Pop()
if r ∉ U and deg r = 2 then
    (a, b) ← neighbors(r)
    Delete vertex r and edges (a, r), (b, r) from the tree S
    Add edge (a, b) to the tree S
end if
```

Algoritms apskata koka virsotnes $\texttt{EulerIn}[v]$ augošā secībā. Algoritms uztur steku ar visām virsotnēm, kuram jau tika uzsākta, bet vēl nav pabeigta apstrāde. Steka augšā vienmēr ir pēdējā apskatīta kopas U virsotne, tā tiek apzīmēta ar S_1. Ar S_2 tiek apzīmēta nākamā virsotne uz steka utt. Tiek uzturēts invariants, ka visām virsotnēm u ar $\texttt{EulerOut}[u] < \texttt{EulerIn}[S_1]$ apstrāde jau ir pabeigta, tātad stekā atrodas tikai virsotnes ar $\texttt{EulerIn}[u] \leq \texttt{EulerIn}[S_1] \leq \texttt{EulerOut}[u]$, jeb virsotnes S_1 priekšteči.

Pievienojot virsotni $v \in U$ apstrādei, algoritms pabeidz apstrādi visām virsotnēm u ar $\texttt{EulerIn}[S_1] \leq \texttt{EulerOut}[u] < \texttt{EulerIn}[v]$. 2. zīmējumā tas ir virsotnes zilajā reģionā un virsotnes uz ceļa no S_1 līdz c (ieskaitot S_1, bet neieskaitot c). Zilajā reģionā nav nevienas kopas U virsotnes, tāpēc visas šī reģiona virsotnes var izdzēst atkārtoti izmantojot pirmo operāciju (lapas izdzēšanu), tātad neviena no šīm virsotnēm nepaliks kokā S un algoritms tās virsotnes vienkārši ignorē. Pēc zilā reģiona izdzēšanas, sarkanām virsotnēm uz ceļa no



2. zīm.: Virtuālā koka konstruēšana

S_1 līdz c pakāpe ir 2 un tās nepieder kopai U (citādi tās atrastos stekā), tātad visas šīs sarkanās virsotnes var tikt izdzēstas, atkārtoti izmantojot (2) operāciju. Tātad pēc visām izdzēšanām kokā S no apskatītām virsotnēm u paliks tikai virsotnes S_1 un S_2. Vispārīgā gadījumā algoritmam ir jāpabeidz apstrāde virsotnēm, kas ir steka augšpusē, un kas atrodas zem virsotnes c. Algoritma cikls 3. – 12. rindiņā apstrādā visas šādas virsotnes izņemot pēdējo (piemērā S_2). Pēdējā virsotne tiek apstrādāta atsevišķi 14. – 26. rindiņā.

Pēc algoritma galvenā cikla (2. – 30. rindiņa) nav pabeigta apstrāde virsotnēm u ar $\texttt{EulerOut}[u] \geq \texttt{EulerIn}[S_1]$, kur S_1 ir virsotne steka augšā. Izmantojot tieši tādu pašu spriedumu, ka augstāk, var pamatot, ka ir iespējams izdzēst visas virsotnes, kas neatrodas stekā. Tātad, lai pabeigtu apstrādi, algoritmam ir tikai jāpievieno šķautnes starp steka virsotnēm, kas tiek izdarīts 32. – 36. rindiņā.

Algoritms pievieno kokam S virsotnes, kas nepieder U, tikai 19. rindiņā. Visām šādām virsotnēm c 22. rindiņā uzreiz tiek pievienota pirmā šķautne, vēlāk tiks pievienota vēl viena šķautne virsotnes v virzienā (skatīt 2. zīmējumu), kā arī tiks pievienota vēl viena šķautne "uz augšu", ja c nav koka S sakne (jeb nav augstākā no koka S virsotnēm kokā T). Tātad šādām virsotnēm būs pakāpe vismaz 3, ar vienīgo izņēmumu – koka S sakni, kurai var būt pakāpe 2. Ja iegūtā koka S saknes pakāpe tiešām ir 2 un tā nepieder U, tad to ir iespējams izdzēst, izmantojot (2) operāciju, kas tiek izdarīts 38. – 43. rindiņā. Pēc šis izdzēšanas virsotņu pakāpes nemainās, tāpēc jaunas virsotnes, ko var izdzēst, nevar parādīties. Tas arī pierāda, ka algoritma beigās tiek iegūts koka T virstoņu apakškopas U virtuālais koks.

Pašā sākumā algoritms sakārto kopas U elementus pēc $\texttt{EulerIn}[v]$ augošā secībā, kas prasa

$O(k \log k)$ laiku. Uz katru kopas U virsotni algoritms veic $O(1)$ `stack.Push`$(\cdot)$ operācijas — pa vienai 9., 20., 24. un 29. rindiņā, tātad arī cikli ar steku 2. – 12. un 32. – 36. rindiņā amortizēti strādā laikā $O(k)$. Visām pārējām algoritma operācijām, ieskaitot zemākā kopīgā priekšteča noteikšanu, ir $O(1)$ laika sarežģītība [14, 15]. Tātad kopēja algoritma sarežģītība ir $O(k \log k)$.

3.4. Nogriežņu koks

Nogriežņu koks ir datu struktūra, ar kuras palīdzību var efektīvi veikt vairākas operācijas ar lineāru masīvu izmērā n laikā $O(\log n)$. Vispārīgā gadījumā nogriežņu koku var pielietot arī citos nolūkos, kā arī ar to palīdzību var izpildīt ļoti dažādu veidu operācijas. Šajā nodaļā tiks aprakstītas tikai divu konkrētu veidu operācijas, kas būs nepieciešamas šī darba ietvaros.

Masīvam izmērā n atbilstošais nogriežņu koks ir pilns binārais koks ar n lapām. Katra nogriežņu koka virsotne v atbilst kādam masīva apakšnogrieznim $[l, r]$. Nogriežņu koka saknei atbilst viss masīvs, jeb nogrieznis $[1, n]$. Nogriežņu koka lapām atbilst masīva atsevišķi elementi, jeb nogriežņi $[x, x]$. Ja virsotne v atbilst nogrieznim $[l, r]$ ar $l < r$, tad tās diviem bērniem atbilst attiecīgi nogriežņi $\left[l, \left\lfloor \frac{l+r}{2} \right\rfloor\right]$ un $\left[\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r\right]$.

Teorēma 3.5. *Nogriežņu koka ar n lapām dziļums ir $O(\log n)$.*

Pierādījums. Ja virsotne v atbilst nogrieznim garumā ℓ, tad tās bērni atbilst nogriežņiem garumā $\leq \left\lceil \frac{\ell}{2} \right\rceil$. Tātad virsotnes dziļumā k, jeb attālumā k no koka saknes, atbilst nogriežņiem garumā $\leq \left\lceil \frac{n}{2^k} \right\rceil$. Tātad visas virsotnes dziļumā $k = \lceil \log_2 n \rceil$ ir lapas, un nogriežņu koka dziļums nepārsniedz k. □

Teorēma 3.6. *Katrs masīva elements pieder $O(\log n)$ virsotņu nogriežņiem.*

Pierādījums. Apskatīsim patvaļīgu masīva elementu ar indeksu x. Apskatīsim patvaļīgu dziļumu k un visas koka virsotnes šajā dziļumā, jeb attālumā k no saknes. Pēc nogriežņu koka konstrukcijas, visu apskatīto virsotņu atbilstošu nogriežņu apvienojums ir $[1, n]$, pie tam nekādi divi no šiem nogriežņiem nekrustojas. Tātad dziļumā k eksistē tieši viena virsotne, kurai atbilstošais nogrieznis satur x. Apvienojot šo novērojumu ar teorēmas 3.5 rezultātu, iegūstam prasīto. □

No teorēmas 3.6 seko, ka operācijas, kas izmaina vienu masīva elementu, ietekmē tikai $O(\log n)$ nogriežņu koka virsotnes. Tātad, ja katru ietekmētu nogriežņu koka virsotni ir iespējams apstrādāt laikā $O(1)$, tad viena elementa izmaiņas operācijai kopējā sarežģītība ir $O(\log n)$.

Lemma 3.1. *Katru masīva prefiksu $[1, r] \subseteq [1, n]$ un sufiksu $[l, n] \subseteq [1, n]$ ir iespējams sadalīt $O(\log n)$ nogriežņos, kas atbilst nogriežņu koka virsotnēm.*

Pierādījums. Apskatīsim tikai prefiksa gadījumu, jo sufiksa gadījuma ir simetrisks. Apskatīsim nogriežņu koka sakni un tās bērnus. Apskatīsim divus gadījumus:

1. Ja prefikss $[1, r]$ krustojās ar labajam dēlam atbilstošo nogriezni, tad $[1, r]$ pilnībā satur kreisajam dēlam atbilstošo nogriezni. Tātad var pievienot sadalījumam kreisajam dēlam atbilstošu nogriezni un rekursīvi pāriet uz labo dēlu.

2. Citādi, prefikss $[1, r]$ ir pilnībā iekļauts kreisajam dēlam atbilstošajā nogrieznī un var rekursīvi pāriet uz kreiso dēlu.

Abos gadījumos aprakstītā procedūra pievieno sadalījumam ne vairāk kā vienu nogriezni un pāriet uz nogriežņu koka virsotni vienu dziļuma līmeni zemāk. Tātad no katra dziļuma līmeņa tiek izvēlēts ne vairāk kā viens nogrieznis un, apvienojot šo ar teorēmas 3.5 rezultātu, iegūstam prasīto. □

Teorēma 3.7. *Katru masīva segmentu $[l, r] \subseteq [1, n]$ ir iespējams sadalīt $O(\log n)$ nogriežņos, kas atbilst nogriežņu koka virsotnēm.*

Pierādījums. Līdzīgi lemmas 3.1 pierādījumam, apskatīsim nogriežņu koka sakni un tās bērnus. Apskatīsim divus gadījumus:

1. Ja segments $[l, r]$ krustojas ar abu dēlu atbilstošajiem nogriežņiem, tad kreisajam dēlam šīs šķēlums ir tām atbilstošā nogriežņa sufikss, bet labajam dēlam — prefikss. Apvienojot šo novērojumu ar lemmas 3.1 rezultātu, iegūstam prasīto.

2. Citādi, segments $[l, r]$ ir pilnībā iekļauts vienā no bērniem atbilstošajiem nogriežņiem un var induktīvi pāriet uz attiecīgo bērnu. □

No teorēmas 3.7 seko, ka operācijas uz patvaļīgiem segmentiem $[l, r] \subseteq [1, n]$ var izpildīt laikā $O(\log n)$, ja katrai nogriežņu koka virsotnei ir iespējams veikt šo pašu operāciju uz virsotnei atbilstošā nogriežņa laikā $O(1)$. Piemēram, ja operācija ir maksimuma atrašana uz segmenta, tad katrā nogriežņu koka virsotnē ir jāglabā maksimums tai atbilstošajā nogrieznī, kas ļauj operāciju "izrēķināt maksimumu" veikt laikā $O(1)$, atgriežot saglabātu vērtību.

4. Vienkāršotais uzdevums

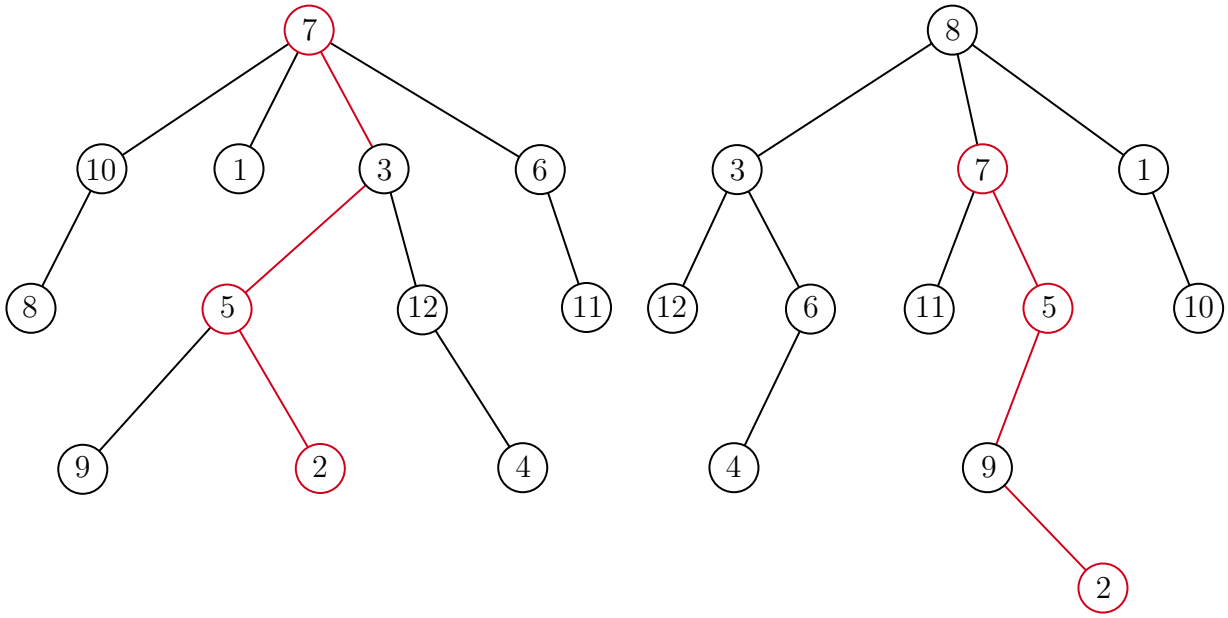
Lai atrisinātu iepriekš aprakstīto uzdevumu, sākumā aprakstīsim algoritmu vienkāršākam uzdevumam, un tad izmantosim to, lai izveidotu algoritmu sākotnējam uzdevumam.

Apskatīsim sakņotu koku T. Virsotņu virkni $v_1, v_2, \ldots, v_k$ sauksim par koka T *dilstošu apakšvirkni*, ja visiem $1 \leq i < k$ virsotne v_{i+1} atrodas stingri iekšā virsotnes v_i apakškokā.

Dotiem diviem sakņotiem kokiem T_1 un T_2 ar kopīgu virsotņu kopu (bet, iespējams, ar dažādām saknēm) katrai virsotnei v ir jāatrod garākās kopīgās dilstošas apakšvirknes ar $v_1 = v$ garumu. Apzīmēsim šīs vērtības ar $f(v)$.

Tālāk parādīsim vienu piemēru un tad aprakstīsim algoritmu šim uzdevumam. 4.2. nodaļā aprakstīsim nepieciešamo palīg datu struktūru, un 4.3. nodaļā aprakstīsim pašu algoritmu.

4.1. Piemērs



3. zīm.: Divu sakņotu koku kopīgas dilstošas apakšvirknes piemērs

3. zīmējumā ir attēloti divi koki ar 12 virsotnēm un saknēm attiecīgi virsotnēs 7 un 8. Ar sarkano krāsu ir iezīmēta šo divu koku kopīgā dilstoša apakšvirkne $7, 5, 2$. Starp visām kopīgajām dilstošām apakšvirknēm ar $v_1 = 7$, šī ir visgarākā, tāpēc $f(7) = 3$.

Piemērā, virsotnēm 3 un 5, $f(3) = f(5) = 2$ un attiecīgas kopīgās dilstošās apakšvirknes ir $3, 4$ (vai $3, 12$) un $5, 2$ (vai $5, 9$). Visām pārējām virsotnēm v, $f(v) = 1$, jo virsotnes v apakškokiem pirmajā un otrajā kokā nav kopīgu virsotņu, izņemot pašu v.

Virkne $10, 8$ ir tikai pirmā koka dilstoša apakšvirkne, savukārt $8, 10$ ir tikai otrā koka dilstoša apakšvirkne.

4.2. Datu struktūra

Šajā nodaļā tiks aprakstīta datu struktūra, kas ļauj izpildīt sekojošās operācijas:

1. Pievienot pieprasījumu (l, r) ar $1 \leq l \leq r \leq n$ struktūrai. Sākotnēji pievienotam pieprasījumam ir piesaistīta vērtība 0;
2. Dotiem x $(1 \leq x \leq n)$ un v, visiem pieprasījumiem, kas pašlaik ir pievienoti struktūrai un kuriem $l \leq x \leq r$, aizvietot tām piesaistītu vērtību a ar $\max(a, v)$;
3. Izņemt pieprasījumu no struktūras. Tiek izņemts pieprasījums, kas bija pievienots struktūrai pēdējais no visiem pašlaik pievienotiem pieprasījumiem. Izņemot pieprasījumu, tas tiek izdzēsts no struktūras un funkcija atgriež tam piesaistīto vērtību;

Aprakstītā datu struktūra spēj apstrādāt katru no šādām operācijām $O(\log n)$ laikā.

4.2.1. Definīcija un inicializācija

Datu struktūra sastāvēs no divām daļām – steka `QueryStack`, kas satur visus pievienotus pieprasījumus, un nogriežņu koka ar n lapām. Katra nogriežņu koka virsotne saturēs atsevišķu steku ar visiem pieprasījumiem, kas pārklāj šis virsotnes segmentu, bet nepārklāj virsotnes vecāka segmentu. Datu struktūra tiek inicializēta laikā $O(n)$ sekojošā veidā:

Algoritms 3 Pieprasījumu nogriežņu koks

```
1:  function InitQuerySegmentTree(n)
2:      QueryStack ← empty stack
3:      QueryTreeRoot ← CreateQueryTree(1, n)
4:  end function
5:
6:  function CreateQueryTree(l, r)
7:      vertex ← new vertex
8:      vertex.L ← l
9:      vertex.R ← r
10:     vertex.queries ← empty stack
11:     vertex.queries.Push(0)
12:
13:     if l < r then
14:         c ← ⌊(l+r)/2⌋
15:         vertex.LeftSon ← CreateQueryTree(l, c)
16:         vertex.RightSon ← CreateQueryTree(c + 1, r)
17:     end if
18:
19:     return vertex
20: end function
```

4.2.2. Pirmā operācija

Pievienojot pieprasījumu, tā segments tiek sadalīts $O(\log n)$ nogriežņos un pieprasījums tiek pievienots attiecīgo nogriežņu koka virsotņu stekiem.

```
21: function PushQuery(l, r)
22:     QueryStack.Push(l, r)
23:     TreePushQuery(QueryTreeRoot, l, r)
24: end function
25:
26: function TreePushQuery(vertex, l, r)
27:     if l ≤ vertex.L and vertex.R ≤ r then
28:         vertex.queries.Push(0)
29:     else
30:         if vertex.LeftSon.R ≥ l then
31:             TreePushQuery(vertex.LeftSon, l, r)
32:         end if
33:         if vertex.RightSon.L ≤ r then
34:             TreePushQuery(vertex.RightSon, l, r)
35:         end if
36:     end if
37: end function
```

4.2.3. Otrā operācija

Otrās operācijas laikā tiek apskatītas visas nogriežņu koka virsotnes, kuru atbilstošajiem nogriežņiem pieder x, un mērķis ir atjaunot piesaistītas vērtības visiem pieprasījumiem, kas atrodas apskatīto virsotņu stekos. Pašas operācijas laikā tiek atjaunotas tikai vērtības steku augšpusēs (viena vērtība katrā stekā, jeb virsotnē), kas vēlāk trešās operācijas laikā tiks slinki izplatītas uz steku zemākiem elementiem (paņēmiens, ko angliski sauc par *lazy propagation*).

```
38: function UpdateQueries(x, v)
39:     TreeUpdateQueries(QueryTreeRoot, x, v)
40: end function
41:
42: function TreeUpdateQueries(vertex, x, v)
43:     a ← vertex.queries.Pop()
44:     vertex.queries.Push(max(a, v))
45:     if vertex.LeftSon exists and vertex.LeftSon.R ≥ x then
46:         TreeUpdateQueries(vertex.LeftSon, x, v)
47:     else if vertex.RightSon exists and vertex.RightSon.L ≤ x then
48:         TreeUpdateQueries(vertex.RightSon, x, v)
49:     end if
50: end function
```

4.2.4. Trešā operācija

Pieprasījuma izņemšanas operācija notiek simetriski pieprasījuma pievienošanai. Tiek apskatītas tas pašas $O(\log n)$ virsotnes kā pieprasījuma pievienošanas laikā. Katrai šādai virsotnei tiek izņemts steka augšējais elements a. Šī vērtība tiek slinki izplatīta un steka nākamo elementu, un kā pieprasījuma rezultāts tiek atgriezta maksimālā no izņemtajām a vērtībām.

```
51: function PopQuery()
52:     (l, r) ← QueryStack.Pop()
53:     return TreePopQuery(QueryTreeRoot, l, r)
54: end function
55:
56: function TreePopQuery(vertex, l, r)
57:     if l ≤ vertex.L and vertex.R ≤ r then
58:         a ← vertex.queries.Pop()
59:         b ← vertex.queries.Pop()
60:         vertex.queries.Push(max(a, b))
61:         return a
62:     else
63:         a ← 0
64:         if vertex.LeftSon.R ≥ l then
65:             a ← max(a, TreePopQuery(vertex.LeftSon, l, r))
66:         end if
67:         if vertex.RightSon.L ≤ r then
68:             a ← max(a, TreePopQuery(vertex.RightSon, l, r))
69:         end if
70:         return a
71:     end if
72: end function
```

Algoritma 59. un 60. rindiņā notiek iepriekš minētā slinkā izplatīšana. Apskatot konkrētu virsotni, tās dzīves ciklu var aprakstīt sekojošā veidā:

1. Pieprasījums b ir pievienots stekam (pirmā operācija);
2. Pieprasījums a ir pievienots stekam (pirmā operācija);
3. Vairākas otrās operācijas;
4. Pieprasījums a ir izņemts no steka (trešā operācija).

Izņemot pieprasījumu a no steka 4. solī, tā vērtība ir vienāda ar maksimumu pāri visām otrajām operācijām 3. solī. Šīs otrās operācija vēl netika pielietotas pieprasījumam b, jo otrās operācijas laikā tā tika pielietota tikai steka augšējām elementam, jeb a. No citas puses, pēc definīcijas, visām otrajām operācijām 3. solī ir jāietekmē arī pieprasījums b. Tātad algoritma 59. un 60. rindiņā visas 3. soļa operācijas tiek pielietotas pieprasījumam b.

4.3. Algoritms

Sekojošs algoritms atrisina vienkāršoto uzdevumu laikā $O(n \log n)$, kur n ir virsotņu skaits.

Sākumā laikā $O(n)$ apstaigā koku T_1 sākot no tā saknes un sagatavo masīvus $\texttt{EulerIn}_1$ un $\texttt{EulerOut}_1$, kas apraksta koka T_1 Eilera apstaigu. Tad inicializē nodaļā 4.2. aprakstītu datu struktūru izmērā n un izsauc zemāk definētu CalcDownDFS funkciju uz koka T_2, sākot no tā saknes.

Algoritms 4 Vienkāršotā uzdevuma risinājums

```
1: function CalcDownDFS(v)
2:     PushQuery(EulerIn_1[v], EulerOut_1[v])
3:     for u ∈ children_2(v) do
4:         CalcDownDFS(u)
5:     end for
6:     dp[v] ← PopQuery() + 1
7:     UpdateQueries(EulerIn_1[v], dp[v])
8: end function
```

Pēc funkcijas CalcDownDFS izpildes masīvs `dp` būs aizpildīts ar prasītām vērtībām, jeb $\texttt{dp}[v]$ būs vienāds ar $f(v)$.

Inicializācijas solī algoritms patērē $O(n)$ laiku. Funkcijas CalcDownDFS izpildes laikā uz katru koka virsotni v tiek pa vienai reizei izsauktas funkcijas PushQuery, PopQuery un UpdateQueries, katra no kurām izpildās laikā $O(\log n)$. Tātad funkcijas CalcDownDFS un arī visa algoritma kopējais izpildes laiks ir $O(n \log n)$.

Lemma 4.1. *$f(v) = \max_u(f(u)) + 1$, kur maksimums tiek ņemts pāri visām virsotnēm u, kas atrodas stingri iekšā virsotnes v apakškokā vienlaicīgi gan kokā T_1, gan kokā T_2.*

Pierādījums. Ar u tiek pārlasīta dilstošas apakšvirknes virsotne v_2. □

Teorēma 4.1. *Aprakstītais algoritms ir korekts.*

Pierādījums. No 4.1. lemmas seko, ka pietiek parādīt, ka PopQuery izsaukums algoritma 6. rindiņā atgriež precīzi $\max_u(\texttt{dp}[u])$ kā lemmā 4.1. Starp algoritma 2. un 6. rindiņām tiek apmeklētas visas tādas un tikai tādas virsotnes, kas atrodas virsotnes v apakškokā kokā T_2, tātad UpdateQueries funkcijas izsaukumi tiek veikti visām tādām un tikai tādām virsotnēm. Pēc datu struktūras definīcijas $\text{UpdateQueries}(\texttt{EulerIn}_1[u], \texttt{dp}[u])$ izsaukums ietekmēs virsotnes v pieprasījumu tad un tikai tad, kad $\texttt{EulerIn}_1[v] \leq \texttt{EulerIn}_1[u] \leq \texttt{EulerOut}_1[v]$, kas ir ekvivalents ar "virsotne u atrodas virsotnes v apakškokā kokā T_1". □

5. Pilnā uzdevuma risinājums

Pirms aprakstīt pašu algoritmu, pierādīsim pāris apgalvojumus, uz kuriem tiks balstīts risinājums.

Apskatīsim koku T un tā centroīdu dekompozīciju $\texttt{CD}(T)$. Apzīmēsim ar $\texttt{SS}(T)$ kopu ar visām koka T apakšvirknēm un ar $\texttt{SS}_v(T)$ kopu ar visām koka T apakšvirknēm, kas iet cauri virsotnei v, jeb kurām izpildās papildus nosacījums $d(v_1, v_k) = d(v_1, v) + d(v, v_k)$.

Teorēma 5.1. $\bigcup\limits_{v \in T} \mathit{SS}_v(\mathit{CC}(v)) = \mathit{SS}(T)$.

Pierādījums. Apskatīsim patvaļīgu $\bar{v} = (\bar{v}_1, \bar{v}_2, \ldots, \bar{v}_k) \in \texttt{SS}(T)$. Ievērosim, ka $\bar{v} \in \texttt{SS}(\texttt{CC}(v)) \iff \{\bar{v}_1, \bar{v}_k\} \subseteq V(\texttt{CC}(v))$, jo $\texttt{CC}(v)$ ir saistīts grafs un visas virsotnes $\bar{v}_i$ atrodas uz ceļa starp $\bar{v}_1$ un $\bar{v}_k$. Tātad pietiek parādīt, ka eksistē v, ka $\{\bar{v}_1, \bar{v}_k\} \subseteq V(\texttt{CC}(v))$ un $d(\bar{v}_1, \bar{v}_k) = d(\bar{v}_1, v) + d(v, \bar{v}_k)$. Apskatīsim $\texttt{CD}(T)$ sakni r un tās bērnus $c_1, c_2, \ldots, c_m$. Ja $\bar{v}_1 = r$ vai $\bar{v}_k = r$, tad $v = r$ apmierina prasīto. Citādi, eksistē tieši viens i, ka $\bar{v}_1 \in V(\texttt{CC}(c_i))$ un tieši viens j, ka $\bar{v}_k \in V(\texttt{CC}(c_j))$. Ja $i \neq j$, tad atkal $v = r$ apmierina prasīto, jo r atrodas uz ceļa starp $\bar{v}_1$ un $\bar{v}_k$. Citādi var induktīvi aizstāt koku T ar $\texttt{CC}(c_i)$. Šis process ir galīgs, jo katru reizi tiek samazināts koka T izmērs, tātad beigās tiks atrasta virsotne v, kas apmierina prasīto. □

No teorēmas 5.1 seko, ka pielietojot centroīdu dekompozīciju kokam T, katrai komponentei $\texttt{CC}(v)$ pietiek apskatīt tikai apakšvirknes, kas pieder $\texttt{SS}_v(\texttt{CC}(v))$, un kopā tiks apskatītas visas koka T apakšvirknes.

Apskatīsim koku T un tā patvaļīgu virsotņu apakškopu $U \subseteq V(T)$. Apzīmēsim ar $\texttt{SS}^U(T)$ kopu ar visām koka T apakšvirknēm $\bar{v} = (\bar{v}_1, \bar{v}_2, \ldots, \bar{v}_m)$, ka $\bar{v}_i \in U \; \forall \, 1 \leq i \leq m$. Apzīmēsim ar T^U koka T virsotņu apakškopas U virtuālo koku.

Teorēma 5.2. $\mathit{SS}^U(T) = \mathit{SS}^U(T^U)$.

Pierādījums. Koks T^U tiek iegūts no T pielietojot operācijas (1) izdzēst lapu un (2) izdzēst virsotni ar pakāpi 2 un pievienot šķautni starp tās kaimiņiem. Parādīsim, ka, ja izdzēstā virsotne nepieder U, tad šādas operācijas nemaina kopu $\texttt{SS}^U(T)$. Lapas izdzēšana nemaina attālumus starp pārējām virsotnēm, tāpēc arī apakšvirknes definīciju neietekmē. Pielietojot otro operāciju attālums starp virsotnēm u un v samazinās par 1 tad un tikai tad, kad izdzēstā virsotne w atradās uz ceļa starp u un v. Apskatot patvaļīgu apakšvirkni $\bar{v} = (\bar{v}_1, \bar{v}_2, \ldots, \bar{v}_m)$, ja w atrodas uz ceļa starp $\bar{v}_1$ un $\bar{v}_m$, tad w arī atrodas uz tieši viena ceļa starp $\bar{v}_i$ un $\bar{v}_{i+1}$, jo visu šo ceļu apvienojums ir precīzi ceļš starp $\bar{v}_1$ un $\bar{v}_m$. Tātad, apzīmējot ar T^* koku pēc otrās operācijas, $\texttt{SS}^U(T) \subseteq \texttt{SS}^U(T^*)$, apskatot izmaiņu no koka T uz T^* un $\texttt{SS}^U(T^*) \subseteq \texttt{SS}^U(T)$, apskatot izmaiņu pretējā virzienā. □

5.1. Algoritms

Balstoties uz teorēmām 5.1 un 5.2, aprakstīsim algoritma pirmo daļu. Ar $\mathtt{SS}_a^B(T)$ tiks apzīmēta kopa $\mathtt{SS}_a(T) \cap \mathtt{SS}^B(T)$.

Sākumā sagatavo abus kokus T_1 un T_2 inducētu apakškoku ātrai konstruēšanai, neatkarīgi veicot katram kokam nepieciešamo priekšapstrādi laikā $O(n)$, kur n ir abu koku virsotņu skaits. Tad pielieto centroīdu dekompozīciju kokam T_1. Dekompozīcijas $\mathtt{CD}_1(T_1)$ komponentes $\mathtt{CC}_1(v)$ tiks apskatītas neatkarīgi un katrai tiks apskatītas tikai apakšvirknes, kas pieder kopai $\mathtt{SS}_v(\mathtt{CC}_1(v))$.

Apstrādājot vienu $\mathtt{CC}_1(v)$ komponenti, tiek uzkonstruēts koka T_2 virsotņu apakškopas $V(\mathtt{CC}_1(v))$ virtuālais koks, jeb $T_2^{V(\mathtt{CC}_1(v))}$. Tad tiek pielietota centroīdu dekompozīcija kokam $T_2^{V(\mathtt{CC}_1(v))}$ un atkal tās komponentes $\mathtt{CC}_2(u)$ tiek apstrādātas neatkarīgi, katrai apskatot tikai apakšvirknes, kas pieder kopai $\mathtt{SS}_u(\mathtt{CC}_2(u))$.

Šādā veidā uzdevums tiek reducēts uz $O(n \log n)$ apakšuzdevumiem ar kopējo $\mathtt{CC}_2(u)$ izmēru $O(n \log^2 n)$. Katrā apakšuzdevumā mērķis ir atrast

$$\max(|\bar{v}| : \bar{v} \in \mathtt{SS}_v(\mathtt{CC}_1(v)) \cap \mathtt{SS}_u(\mathtt{CC}_2(u))),$$

kur $|\bar{v}|$ ir apakšvirknes $\bar{v}$ garums. Ievērojot, ka

$$\mathtt{SS}_v(\mathtt{CC}_1(v)) \cap \mathtt{SS}_u(\mathtt{CC}_2(u))) \subseteq \mathtt{SS}_v^{V(\mathtt{CC}_2(u))}(\mathtt{CC}_1(v)),$$

un pielietojot teorēmas 5.2 rezultātu, iegūstam, ka meklētais maksimums sakrīt ar

$$\max\left(|\bar{v}| : \bar{v} \in \mathtt{SS}_v\left(T_1^{V(\mathtt{CC}_2(u))\cup\{v\}}\right) \cap \mathtt{SS}_u(\mathtt{CC}_2(u))\right).$$

Piezīme. *Koka $T_1^{V(\mathtt{CC}_2(u))\cup\{v\}}$ virsotņu kopai tiek pievienota virsotne v, lai varētu izmantot iepriekšēju kopas $\mathtt{SS}_v(\cdot)$ definīciju.*

Apzīmējot $m = |V(\mathtt{CC}_2(u))|$, abu koku $T_1^{V(\mathtt{CC}_2(u))\cup\{v\}}$ un $\mathtt{CC}_2(u)$ izmēri ir $O(m)$, tāpēc atrisinot katru apakšuzdevumu laikā $O(m \log n)$, tiks iegūts algoritms ar kopējo laika sarežģītību $O(n \log^3 n)$.

Apskatīsim vienu šādu apakšuzdevumu. Tā divus kokus apzīmēsim ar $R_1 := T_1^{V(\mathtt{CC}_2(u))\cup\{v\}}$ un $R_2 := \mathtt{CC}_2(u)$. Kā koka R_1 sakni izvēlēsimies virsotni $r_1 := v$, kokam R_2 — virsotni $r_2 := u$. Apakšuzdevuma mērķis tātad ir atrast

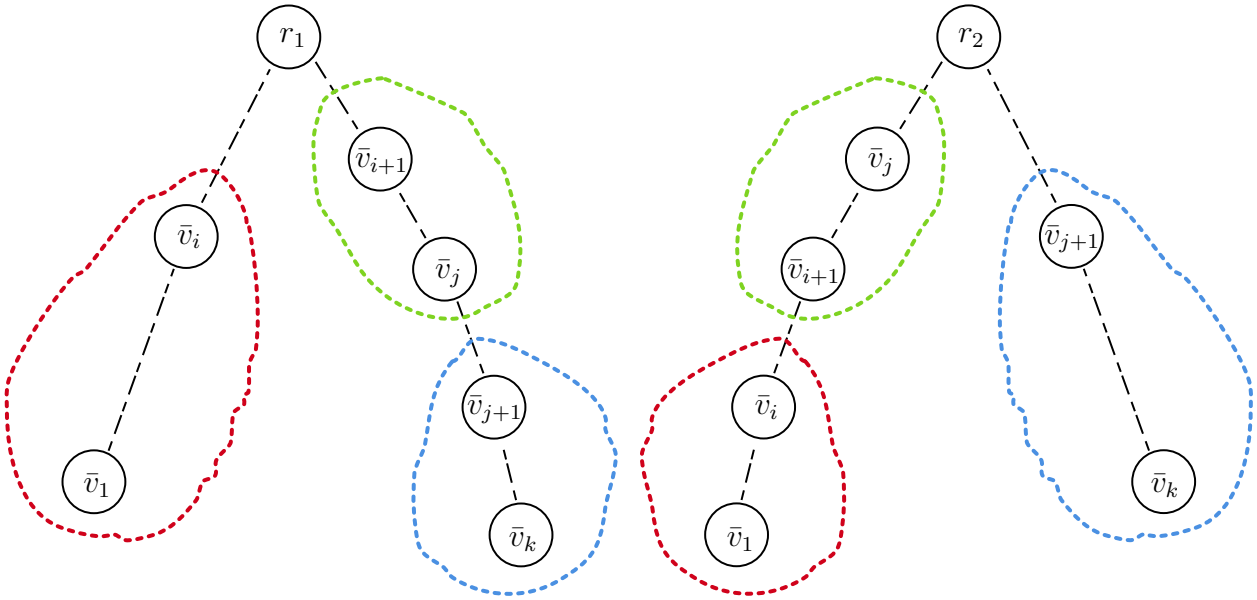
$$\max\left(|\bar{v}| : \bar{v} \in \mathtt{SS}_{r_1}(R_1) \cap \mathtt{SS}_{r_2}(R_2)\right)$$

Turpmāk apakšuzdevuma risinājuma aprakstā izmantosim tikai R_1, R_2, r_1 un r_1 apzīmējumus un neatsauksimies uz oriģinālajiem kokiem T_1 un T_2 vai to centroīdu dekompozīcijām. Vēl

joprojām ar m tiks apzīmēts abu koku izmērs, jeb $|V(R_1)| \in O(m)$ un $|V(R_2)| \in O(m)$.

Ar $P_k(u, v)$ apzīmēsim ceļu no virsotnes u līdz virsotnei v kokā R_k, jeb $P_k(u, v) = (p_1, p_2, \ldots, p_\ell)$, kur $p_1 = u$, $p_\ell = v$, kokā R_k ir šķautne starp p_i un p_{i+1} visiem $1 \leq i < \ell$ un $p_i \neq p_j$ visiem $i \neq j$.

Tagad apskatīsim patvaļīgu $\bar{v} = (\bar{v}_1, \bar{v}_2, \ldots, \bar{v}_k) \in \mathtt{SS}_{r_1}(R_1) \cap \mathtt{SS}_{r_2}(R_2)$. Ar $\bar{v}_i$ apzīmēsim pēdējo no $\bar{v}$ virsotnēm, kas pieder ceļam $P_1(\bar{v}_1, r_1)$. Līdzīgi ar $\bar{v}_j$ apzīmēsim pēdējo no $\bar{v}$ virsotnēm, kas pieder ceļam $P_2(\bar{v}_1, r_2)$. Nezaudējot vispārīgumu, pieņemsim, ka $j \geq i$. 4. zīmējumā ir grafiski attēlota apakšvirkne $\bar{v}$ kokos R_1 un R_2.



4. zīm.: Koku R_1 un R_2 kopīga apakšvirkne

Apakšvirkni $\bar{v}$ var sadalīt trīs daļās, kas 4. zīmējumā ir attēlotas ar sarkanu, zaļu un zilu krāsām. Sarkanā daļa atbilst kādai koku R_1 un R_2 kopīgajai dilstošai apakšvirknei, kas sākās virsotnē $\bar{v}_i$. Līdzīgi zilā daļa atbilst kādai koku R_1 un R_2 kopīgajai dilstošai apakšvirknei, kas sākās virsotnē $\bar{v}_{j+1}$. Savukārt zaļā daļa ir kāda ceļu $P_1(r_1, \bar{v}_{j+1})$ un $P_2(\bar{v}_i, r_2)$ kopīgā apakšvirkne.

Apskatot konkrētas virsotnes $\bar{v}_i$ un $\bar{v}_{j+1}$, lai iegūtu garāko koku R_1 un R_2 kopīgo apakšvirkni ar apskatītām $\bar{v}_i$ un $\bar{v}_{j+1}$ virsotnēm, ir neatkarīgi jāmaksimizē garumi visām trim apakšvirknes daļām. Tātad, izmantojot $\mathtt{dp}[v]$ apzīmējumu no 4.3. nodaļas, un apzīmējot ar $g(u, v)$ ceļu $P_1(r_1, v)$ un $P_2(u, r_2)$ garākās kopīgās apakšvirknes garumu, iegūstam, ka prasītais maksimums ir vienāds ar

$$\max_{\substack{\{u,v\} \in V(R_1) \cap V(R_2) \\ r_1 \in P_1(u,v) \\ r_2 \in P_2(u,v)}} (\mathtt{dp}[u] + g(u, v) + \mathtt{dp}[v]) \,,$$

pārlasot virsotnes $\bar{v}_i = u$ un $\bar{v}_{j+1} = v$.

Tagad aprakstīsim algoritmu, kas izrēķina pēdējo maksimumu laikā $O(m \log m)$. Sākumā laikā $O(m \log m)$, izmantojot 4.3. nodaļā aprakstīto algoritmu, izrēķina vērtības $\texttt{dp}[v]$. Kokam R_2 laikā $O(m)$ veic Eilera apstaigu un aizpilda masīvus $\texttt{EulerIn}_2[v]$, $\texttt{EulerOut}_2[v]$ un $\texttt{SubRoot}_2[v]$, kur $\texttt{SubRoot}_2[v] = p_2$ ir otrā virsotne uz ceļa $P_2(r_2, v) = (p_1, p_2, \ldots, p_\ell)$. Koka R_2 saknei definējam $\texttt{SubRoot}_2[r_2] := r_2$. Masīva $\texttt{SubRoot}$ vērtības tiks izmantotas, lai garantētu, ka $r_2 \in P_2(u, v)$, jo $r_2 \in P_2(u, v) \iff (\texttt{SubRoot}_2[u] \neq \texttt{SubRoot}_2[v] \vee u = v = r_2)$. Papildus sagatavo vēl vienu palīgmasīvu — $\texttt{Euler}_2[\texttt{EulerIn}_2[v]] = v$.

Algoritms apstaiga koku R_1 un pieņem, ka virsotne, kura pašlaik tiek apstrādāta, ir $v = \bar{v}_{j+1}$. Algoritms uztur $\texttt{dp}[u] + g(u, v)$ vērtības visām potenciālām u virsotnēm nogriežņu kokā. Precīzāk, katrai virsotnei $w = \bar{v}_j$ nogriežņu koka atbilstošajā elementā — ar indeksu $\texttt{EulerIn}_2[w]$ — tiek glabāts maksimālais kopējais apakšvirknes garums 4. zīmējuma sarkanai un zaļajai daļām, jeb maksimālā j vērtība. Pirms algoritms sāks koka R_1 apstaigu no tā saknes r_1, tiek uzskatīts, ka ceļš $P_1(r_1, v)$ ir tukšs un tātad arī 4. zīmējuma zaļā daļa ir tukša, kas nozīmē, ka nogriežņu koks ir jāinicializē, virsotnei w izmantojot $\texttt{dp}[w]$ vērtību. Inicializācijas brīdī virsotne w atbilst apakšvirknes virsotnei $\bar{v}_i$.

Algoritma apraksta tiks izmantoti sekojoši apzīmējumi: $\texttt{subtree}_1(v)$ ir virsotnes v apakškoks kokā R_1; $\texttt{children}_1(v)$ ir virsotnes v bērnu kopa kokā R_1; funkcijas SetValue, GetMax un GetMaxPosition atbilst operācijām ar nogriežņu koku. SetValue(a, x) piešķir elementam ar indeksu a vērtību x. GetMax(l, r) atgriež maksimālu vērtību segmentā $[l, r]$. GetMaxPosition(l, r) atgriež indeksu, kurā atrodas kāds no segmenta $[l, r]$ maksimumiem. Precīzs algoritms ir definēts zemāk.

Algoritms 5 Apakšuzdevuma risinājums

```
 1: result ← 0
 2: FirstSubtree ← r_2
 3: FirstRootMax ← −∞
 4: SecondRootMax ← −∞
 5: for w ∈ V(R_1) ∩ V(R_2) do
 6:     SetValue(EulerIn_2[w], dp[w])
 7: end for
 8:
 9: for c ∈ children_1(r_1) do
10:     for u ∈ subtree_1(c) ∩ V(R_2) do
11:         SetValue(EulerIn_2[u], 0)
12:     end for
13:     AddVertex(r_1)
14:     FinalDFS(c)
15:     RemVertex(r_1)
16:     for u ∈ subtree_1(c) ∩ V(R_2) do
17:         SetValue(EulerIn_2[u], dp[u])
18:     end for
19: end for
```

Algoritma cikls 5. – 7. rindiņā inicializē nogriežņu koku. Tālāk ciklā 9. – 19. rindiņā tiek neatkarīgi apskatīti koka R_1 saknes r_1 bērni un to apakškoki. Katram saknes bērnam c, pirmkārt, 10. – 12. rindiņā no nogriežņu koka tiek izdzēstas vērtības, kas atbilst virsotnēm u, kas kokā R_1 atrodas virsotnes c apakškoka. Šis virsotnes nav derīgi kandidāti virsotnei $\bar{v}_i = u$, jo tās neapmierinās $r_1 \in P_1(u, v)$ nosacījumu ($v \in \texttt{subtree}_1(c)$). Funkcijas AddVertex un RemVertex attiecīgi pievieno virsotni ceļa $P_1(r_1, v)$ beigās un izdzēš šī paša ceļa pēdējo virsotni. Šis funkcijas tiks precīzi definētas zemāk, bet to mērķis ir atbilstoši atjaunot nogriežņu koka vērtības, jo izmainoties ceļam $P_1(r_1, v)$ izmainās arī $g(u, v)$ vērtības, jeb 4. zīmējuma zaļās daļas izmērs. Funkcijas FinalDFS izsaukums 14. rindiņā apstrādā visas virsotnes $v \in \texttt{subtree}_1(c)$. Algoritma 15. – 18. rindiņā nogriežņu koks tiek atgriezts sākotnējā stāvoklī, atceļot 10. – 13. rindiņas efektu, lai sagatavotos nākamas virsotnes c apstrādei.

Mainīgais `result` izpildes beigās saturēs algoritma rezultātu. Mainīgie `FirstSubtree`, `FirstRootMax` un `SecondRootMax` aprakstīs divas lielākās $\texttt{dp}[u] + g(u, v)$ vērtības pie $\bar{v}_j = r_2$, jo virsotnei r_2 ir nepieciešama īpaša apstrāde, kā tiks paskaidrots vēlāk. Tagad definēsim funkciju FinalDFS.

```
20: function FinalDFS(v)
21:     if v ∈ V(R_2) and v ≠ r_2 then
22:         p ← SubRoot_2[v]
23:         bestAny ← max(GetMax(1, EulerIn_2[p] − 1), GetMax(EulerOut_2[p] + 1, ∞))
24:         bestRoot ← FirstRootMax if FirstSubtree ≠ p else SecondRootMax
25:         best ← max(bestAny, bestRoot)
26:         result ← max(best + dp[v], result)
27:     end if
28:
29:     AddVertex(v)
30:
31:     for u ∈ children_1(v) do
32:         FinalDFS(u)
33:     end for
34:
35:     RemVertex(v)
36: end function
```

Algoritma 21. – 27. rindiņā tiek apskatītas visas kopīgas apakšvirknes ar $\bar{v}_{j+1} = v$. Tas tiek izdarīts apskatot visus iespējamus kandidātus virsotnei $\bar{v}_j$. Algoritma 10. – 12. rindiņā jau tika nodrošināts, ka visām nogriežņu koka vērtībām izpildās $r_1 \in P_1(u, v)$ nosacījums. Tātad tagad algoritmam ir tikai jānodrošina, ka izpildās $r_2 \in P_2(u, v)$. Sākumā algoritms apskata visus kandidātus $\bar{v}_j \neq r_2$. Ievērojot, ka $r_2 \in P_2(u, v) \iff \texttt{SubRoot}_2[u] \neq \texttt{SubRoot}_2[v]$ un $\texttt{SubRoot}_2[u] = \texttt{SubRoot}_2[\bar{v}_j]$, secinam, ka šādiem kandidātiem algoritmam ir jānodrošina, ka $\texttt{SubRoot}_2[\bar{v}_j] \neq \texttt{SubRoot}_2[v]$. Izmantojot p definīciju algoritma 22. rindiņā, pēdējais nosacījums ir ekvivalents ar $\texttt{EulerIn}_2[\bar{v}_j] < \texttt{EulerIn}_2[p] \vee \texttt{EulerIn}_2[\bar{v}_j] > \texttt{EulerOut}_2[p]$, tātad

algoritma 23. rindiņa izrēķina precīzi maksimālu $\texttt{dp}[u] + g(u, v)$ vērtību pāri visām $\bar{v}_j \neq r_2$. Gadījumā, ja $\bar{v}_j = r_2$, līdzīga pieeja nestrādā, jo vienādība $\texttt{SubRoot}_2[u] = \texttt{SubRoot}_2[\bar{v}_j]$ vairs nav spēkā. Algoritma laikā tie uzturētas trīs vērtības — `FirstRootMax` ir maksimālā $\texttt{dp}[u] + g(u, v)$ vērtība pie $\bar{v}_j = r_2$, kas tiek sasniegta pie $\texttt{SubRoot}_2[u] = \texttt{FirstSubtree}$; un `SecondRootMax` ir maksimālā $\texttt{dp}[u] + g(u, v)$ vērtība pie nosacījuma, ka $\texttt{SubRoot}_2[u] \neq \texttt{FirstSubtree}$. Izmantojot šis trīs vērtības algoritma 24. rindiņā tiek apstrādāts gadījums $\bar{v}_j = r_2$. Algoritma 25. rindiņā tātad tiek iegūts maksimums pāri visām $\bar{v}_j$ kandidātu virsotnēm, un 26. rindiņā tiek atjaunots algoritma rezultāts izmantojot visas apskatītās apakšvirknes (ar $\bar{v}_{j+1} = v$).

Piezīme. *Pēc definīcijas $\bar{v}_{j+1}$ ir pirmā virsotne uz ceļā $P_2(\bar{v}_1, \bar{v}_k)$, kas atrodas stingri pēc r_2, tāpēc gadījums $\bar{v}_{j+1} = r_2$ nav iespējams.*

Tagad tiks definētas funkcijas AddVertex un RemVertex, kas noslēgs algoritma aprakstu.

```
37: function AddVertex(v)
38:     if v ∈ V(R_2) then
39:         if v = r_2 then
40:             FirstRootMax ← GetMax(1, ∞) + 1
41:             MaxEulerIn_2 ← GetMaxPosition(1, ∞)
42:             MaxVertex ← Euler_2[MaxEulerIn_2]
43:             FirstSubtree ← SubRoot_2[MaxVertex]
44:             MaxBefore ← GetMax(1, EulerIn_2[FirstSubtree] − 1)
45:             MaxAfter ← GetMax(EulerOut_2[FirstSubtree] + 1, ∞)
46:             SecondRootMax ← max(MaxBefore, MaxAfter) + 1
47:         else
48:             best ← GetMax(EulerIn_2[v], EulerOut_2[v])
49:             SetValue(EulerIn_2[v], best + 1)
50:         end if
51:     end if
52: end function
53:
54: function RemVertex(v)
55:     if v ∈ V(R_2) then
56:         if v = r_2 then
57:             FirstRootMax ← −∞
58:             SecondRootMax ← −∞
59:         else
60:             SetValue(EulerIn_2[v], 0)
61:         end if
62:     end if
63: end function
```

Pievienojot virsotni v ceļam $P_1(r_1, v)$, ceļiem $P_1(r_1, v)$ un $P_2(u, r_2)$ parādās jaunas kopīgas

apakšvirknes. Acīmredzot, visām jaunām apakšvirknēm pēdējais elements ir vienāds ar v. Apskatītas apakšvirknes 4. zīmējumā atbilst apakšvirknes prefiksam līdz $\bar{v}_j$, jeb sarkanajai un zaļajai daļām kopā. Tātad algoritmam ir jāatrod garākais prefiks ar $\bar{v}_j = v$. Tas tiek panākts apskatot potenciālās $\bar{v}_{j-1}$ virsotnes. Kokā R_2 virsotnei $\bar{v}_{j-1}$ ir jābūt virsotnes v apakškokā. Kokā R_1 virsotnei $\bar{v}_{j-1}$ ir vai nu jābūt uz ceļa $P_1(r_1, v)$ (kas atbilstu 4. zīmējuma zaļajai daļai), vai nu ceļam $P_1(v, \bar{v}_{j-1})$ ir jāsatur virsotni r_1 (kas atbilstu 4. zīmējuma sarkanajai daļai un tad $j - 1 = i$). Nosacījums par koku R_1 tiek automātiski apmierināts ar nogriežņu koka stāvokli, jo tajā nenulles vērtības ir piešķirtas visām tādām un tikai tādām virsotnēm, kā ir prasīts. Nosacījums par koku R_2 tiek apmierināts veicot pieprasījumu nogriežņu koka attiecīgajam nogrieznim. Kā bija pamatots iepriekš, funkcija AddVertex atsevišķi apskata gadījumus $v = r_2$ un $v \neq r_2$. Funkcijas RemVertex definīcija ir simetriska funkcijai AddVertex — tā piešķirt vērtībām, kas tika nomainītas funkcijā AddVertex, to sākotnējās vērtības.

Apakšuzdevuma risinājuma sākumā tika pieņemts, ka $j \geq i$, tāpēc aprakstītais algoritms apskata tikai apakšvirknes, kas apmierina šo nevienādību.

Teorēma 5.3. *Kopām $\mathtt{SS}_{r_1}(R_1)$ un $\mathtt{SS}_{r_2}(R_2)$ eksistē kopīgā apakšvirkne ar maksimālo garumu, kas apmierina nevienādību $j \geq i$.*

Pierādījums. Apskatīsim patvaļīgu garāko kopas $\mathtt{SS}_{r_1}(R_1) \cap \mathtt{SS}_{r_2}(R_2)$ apakšvirkni $\bar{v} = (\bar{v}_1, \bar{v}_2, \ldots, \bar{v}_k)$. Ja tai izpildās $j \geq i$, tad prasītais ir pierādīts. Citādi, apskatīsim virkni $\tilde{v} = (\bar{v}_k, \bar{v}_{k-1}, \ldots, \bar{v}_1)$. Acīmredzot, $\tilde{v}$ arī pieder abām kopām $\mathtt{SS}_{r_1}(R_1)$ un $\mathtt{SS}_{r_2}(R_2)$. Atceroties, ka i un j ir definēti konkrētai apakšvirknei, ar $i(x)$ un $j(x)$ apzīmēsim atbilstošās vērtības virknei x. Ievērojot, ka $i(\tilde{v}) = k - i(\bar{v})$ un $j(\tilde{v}) = k - j(\bar{v})$, iegūstam, ka $j(\bar{v}) < i(\bar{v}) \implies j(\tilde{v}) > i(\tilde{v})$, tātad apakšvirkne $\tilde{v}$ apmierina prasīto. □

No 5.3. teorēmas seko, ka, arī apskatot tikai apakšvirknes, kas apmierina nevienādību $j \geq i$, algoritms iegūst pareizu rezultātu. Tas pierāda, ka aprakstītais algoritms ir korekts.

Piezīme. *Ja 5.3. teorēma nebūtu patiesa, apakšuzdevuma risinājumam varētu atkārtoti pielietot aprakstīto algoritmu divas reizes, otrajā reizē apmainot kokus R_1 un R_2 vietām.*

Aprakstītajam algoritmam apakšuzdevuma risinājumam ir $O(m \log m)$ laika sarežģītība, jo katrai no $O(m)$ koka R_1 virsotnēm tiek veikts konstants operāciju skaits ar nogriežņu koku, un katra nogriežņu koka operācija tiek izpildīta laikā $O(\log m)$.

Apvienojot aprakstīto algoritmu apakšuzdevumu risinājumam kopā ar pilna uzdevumu algoritma pirmo daļu (centroīdu dekompozīcijām un virtuālo koku konstruēšanu) tiek iegūts algoritms pilnā uzdevuma risinājumam ar laika sarežģītību $O(n \log^3 n)$.

Aprakstītais algoritms kā rezultātu atrod tikai garākās kopīgās apakšvirknes garumu, bet ar nebūtiskām izmaiņām var iegūt algoritmu ar tādu pašu sarežģītību, kas rezultātā atradīs vienu no garākajām kopīgajām apakšvirknēm.

6. Rezultāti un secinājumi

Darba rezultāts ir 5.1. nodaļā aprakstītais algoritms, kas pierāda, ka garākās kopīgās apakšvirknes uzdevuma vispārinājums, kas ir definēts 1. nodaļā, ir risināms laikā $O(n \log^3 n)$.

Darba rezultāts apmierina darba mērķi, jo tika apskatīts vienkāršs un dabisks garākās kopīgās apakšvirknes uzdevuma vispārinājums uz kokiem, kas tika atrisināts zemkvadrātiskā laikā.

Literatūra

[1] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Commun. ACM*, vol. 18, p. 341–343, June 1975.

[2] J. D. Ullman, A. V. Aho, and D. S. Hirschberg, "Bounds on the complexity of the longest common subsequence problem," *J. ACM*, vol. 23, p. 1–12, Jan. 1976.

[3] M. L. Fredman, "On computing the length of longest increasing subsequences," *Discrete Mathematics*, vol. 11, no. 1, pp. 29–35, 1975.

[4] E. Kubicka, G. Kubicki, and F. R. McMorris, "An algorithm to find agreement subtrees," *Journal of Classification*, vol. 12, pp. 91–99, Mar 1995.

[5] M. Steel and T. Warnow, "Kaikoura tree theorems: Computing the maximum agreement subtree," *Information Processing Letters*, vol. 48, no. 2, pp. 77 – 82, 1993.

[6] P. Bille and I. L. Gørtz, "Matching subsequences in trees," in *Algorithms and Complexity* (T. Calamoneri, I. Finocchi, and G. F. Italiano, eds.), (Berlin, Heidelberg), pp. 248–259, Springer Berlin Heidelberg, 2006.

[7] A. Lozanoand and G. Valiente, "On the maximum common embedded subtree problem for ordered trees," 01 2004.

[8] S. Mozes, D. Tsur, O. Weimann, and M. Ziv-Ukelson, "Fast algorithms for computing tree lcs," in *Combinatorial Pattern Matching* (P. Ferragina and G. M. Landau, eds.), (Berlin, Heidelberg), pp. 230–243, Springer Berlin Heidelberg, 2008.

[9] R. Tarjan and U. Vishkin, "Finding biconnected componemts and computing tree functions in logarithmic parallel time," in *25th Annual Symposium onFoundations of Computer Science, 1984.*, pp. 12–20, 1984.

[10] N. Megiddo, A. Tamir, E. Zemel, and R. Chandrasekaran, "An $O(n \log^2 n)$ algorithm for the k-th longest path in a tree with applications to location problems," *SIAM Journal on Computing*, vol. 10, no. 2, pp. 328–337, 1981.

[11] G. N. Frederickson and D. B. Johnson, "Generating and searching sets induced by networks," in *Automata, Languages and Programming* (J. de Bakker and J. van Leeuwen, eds.), (Berlin, Heidelberg), pp. 221–233, Springer Berlin Heidelberg, 1980.

[12] N. Megiddo, "Applying parallel computation algorithms in the design of serial algorithms," *J. ACM*, vol. 30, p. 852–865, Oct. 1983.

[13] C. Jordan, "Sur les assemblages de lignes.," vol. 1869, no. 70, pp. 185–190, 1869.

[14] D. Harel and R. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM J. Comput.*, vol. 13, pp. 338–355, 1984.

[15] B. Schieber and U. Vishkin, "On finding lowest common ancestors: Simplification and parallelization," in *AWOC*, 1988.

Maģistra darbs: Zemkvadrātisks algoritms koku ceļu apakšvirkņu uzdevumam

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: _____
(Autora paraksts)

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto maģistra darbu un atzīstu to par **p i e m ē r o t u / n e p i e m ē r o t u** (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu maģistrantūrā.

Darba vadītājs: _____
(Vadītāja paraksts)

Darbs iesniegts maģistrantūras sekretariātā _____.
(Iesniegšanas datums)

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā.

Studiju metodiķe: _____.
(Metodiķes paraksts)

Recenzents: LU prof., Dr. dat. Juris Vīksna
(Akad. amats, zin. grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____
(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____
(Sekretāra paraksts)