Lösningsförslag lägertävlingen 2019

Nils Gustafsson, Simon Lindholm, Erik Amirell Eklöf

Efterlyst

Låt oss kalla de givna noderna för *speciella* noder.

Delpoäng 1

Låt p vara den minsta (längst till vänster) av de speciella noderna, och låt q vara det största. Nu kommer alla platser r som uppfyller $r \leq p$ eller $r \geq q$ vara möjliga svar, så det är bara att printa ut dem. Se upp med fallet $k = 1$.

Delpoäng 2

Den här delpoängsnivån hjälper inte så mycket för att ta de högre nivåerna, så här är en kortfattad beskrivning:

När grafen är ett träd så spelar inte vikterna på kanterna någon roll eftersom det bara finns en väg mellan varje par av noder. Om vi rotar trädet i en av de speciella noderna så kommer vägens ändpunkter inte ha några andra speciella noder i sitt subträd. Detta gäller i alla fall utom när roten vi valde är en av ändpunkterna.

När vi har rett ut de olika fallen blir svaret alla noder som är i någon av ändpunkternas subträd. Efter det återstår det bara att kolla att svaret är giltigt genom att kolla att alla speciella noder är på vägen mellan ändpunkterna. Jobbigt att implementera, men ger en del poäng.

Delpoäng 3, 4, 5

Här finns det lite olika lösningar som ger olika mycket poäng. En viktig observation som vi kommer behöva är att om

$$b_1 \to b_2 \to \cdots \to b_r$$

är en kortaste väg mellan b_1 och b_r, så är

$$b_i \to \cdots \to b_j$$

en kortaste väg mellan b_i och b_j. Detta innebär att om vi letar efter en kortaste väg till y som går igenom alla speciella noder, så kan vi anta att vägen börjar vid en av de speciella noderna.

Låt oss testa en av de speciella noderna som startpunkt. Gör en Dijkstra från y för att räkna ut avstånden $d(y, i)$ till alla andra noder. De kortaste vägarna bildar en slags riktad acyklisk graf. Vi kan tänka oss att en riktad kant går mellan i och j om $d(y, i) + w_{ij} = d(y, j)$. Nu behöver vi bara hitta alla noder sådana att det finns en väg i den riktade grafen som går igenom alla speciella noder och hamnar på y. Det här går att göra i linjär tid, så totalt blir komplexiteten $O(km \log(n))$.

Delpoäng 6

I förra delpoängsnivån löste vi problemet i $O(km \log(n))$ genom att kolla alla möjliga startpunkter. Den huvudsakliga observationen för att lösa hela problemet är att det bara finns två speciella punkter som vi behöver kolla. Låt oss säga att det finns en kortaste väg som går igenom alla speciella noder i ordningen $a_1, \ldots, a_k$ och slutar på någon nod y. Då är ju avståndet mellan a_i och a_j samma som avståndet på vägen. Detta innebär att a_1 och a_k är de speciella punkter som är längst ifrån varandra. Ändpunkterna är alltså samma för alla möjliga vägar, så vi kan lösa problemet genom att köra $O(km \log(n))$-lösningen fast bara för de två ändpunkterna.

Nu återstår bara att faktiskt hitta ändpunkterna. Det antagligen enklaste sättet att göra detta är att göra en Dijkstra från en godtycklig speciell nod, och se vilken av de andra speciella noderna som är längst bort. Denna nod a_1 måste vara en av ändpunkterna, och då kan vi hitta den andra ändpunkten genom att göra samma sak från a_1. Komplexiteten är $O(m \log(n))$.

1 Poplåtar

Poplåtar är i botten en simuleringsuppgift, som går att lösa olika optimerat med lite olika tidskomplexiteter.

Naiv simuleringslösning

En metod vi skulle kunna tänka oss är följande naiva lösning: loopa över startpunkt och slutpunkt för refrängen. Loopa över storleken av refrängens textrader.

Loopa över hela refrängen och beräkna längden av det delade suffixet. Skriv ut maximum av alla dessa svar.

Det är lätt att tänka att den här lösningen är alldeles för långsam – vi har trots allt (minst) 4 nästlade loopar, som alla går upp till N. Vi kan dock notera att när vi loopar över textradens längd så behöver vi bara loopa över alla tal som refrängens längd är delbar med. Refrängens längd kommer i princip att vara ett slumpmässigt tal mellan 1 och N, och ett slumpmässigt tal kommer att vara delbart med 2 med sannolikhet $1/2$, 3 med sannolikhet $1/3$, o.s.v. I genomsnitt kommer talet därför ha ungefär $1/1 + 1/2 + 1/3 + \cdots + 1/N = O(\log N)$ delare.

Tidskomplexiteten på den här lösningen är därmed $O(N^3 \log N)$. Konstantfaktorn är dessutom liten – vi loopar så att start $<$ slut, och i den innersta loopen loopar vi bara upp till (slut $-$ start). Detta vinner en faktor 6 över vad man naivt kanske tänker sig. Dessutom är mängden arbete som utförs i den innersta loopen väldigt lite – bara en jämförelse av två tecken. $O(N^3 \log N)$ går därför igenom med marginal för $N = 400$ och en 2 sekunders tidsgräns, förhoppningsvis även i Python.

Mindre naiv simuleringslösning

En observation vi skulle kunna göra i ovanstående lösning är att flaskhalsen är att beräkna längder på gemensamma suffix. Dessa suffix har dock en hel del överlapp! Det finns nämligen bara $O(N^2)$ suffix vi möjligen skulle kunna vilja jämföra, och vi skulle kunna förberäkna antal tecken de har gemensamt i $O(N^3)$ tid (eller $O(N^2 \log N)$ med stränghashning + binärsökning, men det behövs inte).

Om vi byter ut den innersta nästlade loopen till att loopa över alla textrader och använda en förberäknad lookup-array för att jämföra intilliggande suffix så uppträder flaskhalsen när textradslängden är 1, och vi reducerar vår körningstid till $O(N^3)$.

Med lite optimeringar går det att få igenom den här lösningen på delgrupp 2.

Smartare lösning

Det som gör ovanstående lösningar långsamma är att de loopar över både start- och slutpunkt, och sen utför en massa jobb. Går det att loopa över bara startpunkt, sen utföra allt jobb, och bara loopa över slutpunkt mot slutet?

Svaret är att ja, det går det. Låt oss till att börja med loopa över alla möjliga textradslängder, och prova att starta från absoluta början av strängen. Om vi loopar över alla textrader kan vi då få ut en array av tal som beskriver vad de gemensamma suffixlängderna är mellan varje par av intilliggande textrader. Vi kan därefter loopa över hur många av dessa textrader vi vill ha med, och

hålla koll på vårt nuvarande minimum. För varje antal textrader har vi ett kandidatsvar som är det antal multiplicerat med det nuvarande minimumet.

För att prova nästa startposition är det stora problemet att vi snabbt måste få ut en ny array av gemensamma suffixlängder. Det finns två sätt vi skulle kunna göra det på: antingen kan vi använda hashning + binärsökning som nämnt ovan, eller så kan vi återanvända vår tidigare array. Om vi flyttar startpositionen ett steg lägger vi ju enbart på ett tecken i varje suffix. Om det tecknet är samma för ett par av intilliggande textrader så ökar deras gemensamma suffixlängd med 1 (om den inte är lika med textradens längd), annars blir den nya gemensamma suffixlängden 0.

Vi kan därefter loopa igenom textraderna för den här nya startpositionen, och upprepa för varje startposition.

Körningstiden för detta ges av

$$\sum_{\text{len} \leq N} \left(N + \sum_{\text{start} \leq N} \frac{N}{\text{len}} \right) = O(N^2 \log N),$$

vilket är tillräckligt för delgrupp 2.

Ännu smartare lösning

Det går att optimera ovanstående ännu lite mer, och få bort logfaktorn. Det vi kan göra är att loopa startpositionen enbart över positionerna $0, 1, \ldots, \text{len} - 1$, och sedan ta ut den bästa delsekvensen av textrader på ett smartare sätt än att bara kolla på prefix av arrayen som beskriver de gemensamma suffixlängderna.

Vi kan formulera vårt delproblem så här: givet en array av tal, hitta det delintervall för vilket minimum av talen i det intervallet multiplicerat med intervallängden +1 är så stor som möjligt.

Detta går att lösa i linjär tid! Det vi vill göra är att för varje tal räkna ut vad det maximala intervallet är som har det här talet som minimum, och det kan vi göra genom en svepning. Vi kan anta WLOG att alla tal i sekvensen är unika (genom att t.ex. hantera elementen som par $(tal, index)$ som sorteras primärt baserat på tal, sekundärt på index).

Vi sveper från vänster till höger, och håller en (ökande) stack av de tal för vilka vi ännu inte sett något mindre tal till höger om det, tillsammans med deras positioner. När vi stöter på ett tal, kolla på översta talet i stacken. Om detta är större än vårt nuvarande tal så betyder det att om vi vill att det talet ska vara minimum, så är det största intervallet vi kan välja det som sträcker sig mellan det näst översta talet i stacken, och vår nuvarande position. Hantera detta som ett potentiellt svar, och poppa därefter stacken. Efter att vårt nuvarande tal är större än det sista talet i stacken (och därmed allt däri), pusha det på stacken.

Om vi initialt lägger till talet $-\infty$ (med position -1) i stacken, och i slutet av sekvensen lägger talet $-\infty$, så kommer efter genomsvepningen alla tal ha poppats från stacken, och fått sina svar beräknade. På så sätt har vi löst delproblemet på linjär tid.

Körningstiden kör den här smartare lösningen ges av

$$\sum_{\text{len} \leq N} \left(N + \sum_{\text{start} \leq \text{len}} \frac{N}{\text{len}} \right) = O(N^2),$$

vilket löser delgrupp 3. (Gränsen för N behövde dessvärre vara ganska hög, för att undvika att optimerade lösningarna till de två första delgrupperna gick igenom även på den sista. Lite konstantfaktoroptimeringar kan därmed krävas.)

Trädreklam

Problemet blir något enklare att implementera om man istället för att göra reklam i kanter (vägar) gör reklam i den nod (stad) som ligger i den bortre änden av varje kant (sett från huvudstaden).

Notera att om man gör reklam i två noder A och B sådana att B är en del av subträdet rotat i A, så kommer reklamen i B inte tillföra någonting eftersom alla personer som åker genom B också kommer åka genom A.

Lösningsideén är att göra en pre-order traversering och sedan en knapsack-liknande DP över denna. DP:n skiljer sig från en vanlig knapsack genom att man i fallet då man väljer att göra reklam "skippar" det subträdet. Detta kan man göra genom att man, istället för att rekursera till nästa element, rekurserar till det element som kommer efter subträdet. I en pre-order traversering kommer alla noder i ett subträd ligga konsekutivt och kan därmed hoppas över om man förberäknar antalet noder i varje subträd.

Likt en knapsack definierar vi vår DP-funktion som: givet ett index i traverseringen (i) och hur många kronor som finns att spendera, vad är det maximala antalet personer från städer till höger om i som får se reklamen om vi sätter upp reklam optimalt bland dessa städer?

Som en rekursiv funktion:

$$f(i,b) = \begin{cases} 0 & i > N \\ f(i+1,b) & b < c_i \\ max(f(i+1,b), f(i+s_i, b-c_i) + p_i) & \end{cases}$$

Där s_i är antalet noder i subträdet med rot i nod i och p_i är det totala antalet personer i samma subträd. DP:n fungerar alltså som en vanlig knapsack förutom att man adderar s_i till i istället för 1 i fallet då man väljer att göra reklam.

Tidskomplexiteten blir $O(NB)$.