Teoriblad 1 – Programmeringsolympiaden 2023/24

Erik Hedin

Mars 2023

Jag hävdar att jag har lösningar som tar följande poäng på uppgifterna:

- 10p på **Problem 1**.
- 10p på **Problem 2**.
- 10p på **Problem 3**, fast med en alternativ lösning för 7p.
- 10p på **Problem 4**.

1 Sweep

Problem 1. Det finns ett rutnät av storleken $N \times N$ där K rutor är blockerade. Du får också koordinaterna för de K rutor som är blockerade. Du ska räkna det totala antalet rektanglar man kan skapa som innehåller det övre vänstra hörnet.

I alla deluppgiften gäller det att $N \leqslant 10^{18}$.

Deluppgift 1.1 (2 poäng). $N^2 + K \log K < 10^9$.

Deluppgift 1.2 (3 poäng). $N + K \log K < 10^9$.

Deluppgift 1.3 (5 poäng). $K \log K < 10^9$.

Lösning Problem 1. Följande lösning löser problemet på $O(K \log K)$. Börja med att sortera koordinaterna av de K blockerade rutorna i ökande ordning i x i första hand, och ökande ordning i y i andra hand. Detta går på tid $O(K \log K)$.

I denna lösning kommer den övre vänstra cellen av rutnätet att ha koordinat $(1, 1)$, att x-koordinaterna ökar högeråt, och att y-koordinaterna ökar nedåt.

Fixera bredden w av rektangeln och låt H_w vara den högsta höjden rektanglar med den bredden kan ha (och innehålla den övre vänstra cellen). Notera $H_w = \min\big(N, \min_{1 \leqslant i \leqslant I(w)}\{y_i\}\big)$, där $I(w)$ är index av den sista blockerade rutan i sorterad ordning som har x-koordinat $\leqslant w$. Det finns då H_w sådana rektanglar.

Notera att det totala antalet rektanglar S är lika med:

$$S = N(x_1 - 1) + (N - x_K + 1)H_{x_K} + \sum_{i=2}^{K}(x_i - x_{i-1})H_{x_{i-1}}.$$

Notera att $H_{x_i} = \min(H_{x_{i-1}}, y_i)$ för $i \geqslant 2$ och $H_{x_1} = y_1$, så alla använda H-värden i summan går att beräkna med dp på $O(K)$. Varje term går sedan att beräkna på $O(1)$, och därmed går S att beräkna på $O(K)$ tid totalt. (Notera att N är begränsad, så aritmetiska operationer går på konstant tid.)

Detta löser problemet fullständigt. □

2 Träd-DP

Problem 2. Du får givet ett träd med N noder numererade från 1 till N. Du vill hitta en ordning på noderna $a_1, a_2, \ldots, a_N$, där $a_1 = 1$ och $a_N = N$, och avståndet mellan a_i och a_{i+1} är som mest 2 för alla $i = 1, 2, \ldots, N - 1$. Avståndet mellan två noder är antalet kanter du besöker för att ta dig från den ena till den andra. Om en sådan ordning finns ska du hitta vilken som helst. Annars ska du rapportera att det inte finns någon ordning som uppfyller kriteriet.

Om inte specifierat gäller det att $N \leqslant 10^6$ för alla deluppgifter.

Deluppgift 2.1 (2 poäng). Graden på varje nod är $\leqslant 3$.

Deluppgift 2.2 (3 poäng). $N \leqslant 1000$.

Deluppgift 2.3 (5 poäng). Inga ytterligare begränsningar.

Lösning Problem 2. Följande lösning är $O(N)$, eller rimligtvis $O(N \log N)$ om man är lat i sin implementering någonstans.

Låt B vara en *fullständig* sekvens av noder $b_1, \ldots, b_M$ så att kanten $\{b_i, b_{i+1}\}$ alltid finns i träden för alla i. Dessutom måste $b_1 (= a_1)$ och $b_M (= a_N)$ vara 'markerade' element av sekvensen B. Om $b_i = a_j$ är ett markerat element måste antingen b_{i+1} eller b_{i+2} också vara markerade (och lika med a_{j+1}) (förutom om $i = M \iff j = N$). Slutligen existerar det ett unikt i för varje j så att $b_i = a_j$ och b_i är markerat. Vi noterar att problemet är ekvivalent med att hitta en sådan fullständig sekvens B, eller avgöra att en sådan sekvens inte existerar. Låt $A = (a_1, \ldots, a_N)$ vara dess naturligt tillhörande permutation av markerade noder.

Om $N = 1$ och $a_1 = a_N$ går problemet att lösa triviellt. Hädanefter kommer vi därför anta att $a_1 \neq a_N \iff N > 1$.

Beteckna vägen från a_1 till a_N med p. Betrakta att 'rota trädet i p', vilket kommer innebära att vi riktar varje kant i trädet som inte är på p i riktningen bort från p. Nedan följer ett lemma, som jag inte kommer att bevisa (hehe...):

- Varje kant $e \notin p$ i trädet kommer att användas exakt två gånger i en fullständig sekvens B, om en sådan existerar.

För varje nod $v \notin p$ i trädet kan vi beräkna $dp[v]$; om det är möjligt att använda kanten $(p[v], v)$, markera alla noder i subträden av v, och sedan lämna med kanten $(v, p[v])$ (där $p[v]$ är föräldern av v i det 'rotade' trädet), med lite olika variationer:

i. v är ett löv,

ii. det går att markera alla noder i subträdet av v genom att från $p[v]$ hoppa över v i A på vägen in och sluta på v på vägen ut,

iii. det går att markera alla noder i subträdet av v genom att markera v på vägen in och hoppa över v i A till $p[v]$ på vägen ut,

Notera att ii. och iii. är ekvivalenta. Notera även att det inte går att markera alla noder i subträdet av v (inlkusive v självt) genom att hoppa över v i A både på vägen in och ut, eftersom man då markerar $p[v]$ två gånger.

Det följer att v kan ha som mest ett barn u som inte bara är löv, eftersom annars kommer v eller $p[v]$ behöva markeras två gånger. Därför har dp:n som beräknar $dp[v]$ för alla v endast följande 'rekursionsformler':

- Om v är ett löv är det uppenbart att i. är möjligt,
- Om v bara har barn som är löv kommer följande sekvens av drag uppfylla ii.: $p[v] \to \text{löv}^{\circlearrowleft} \to v$,
- Om v har exakt ett barn u vars subträd inte är ett löv kommer följande sekvens av drag uppfylla ii.: $p[v] \to (\text{löv}^{\circlearrowleft} \to)\ u \to v$, där vi slänger in att besöka alla löv-barn av u. Vi kräver då såklar såklart att $dp[u]$ har ii. som möjligt tillstånd, annars går det inte,
- Om v har två eller fler barn vars subträd inte är ett löv är ingendera av dp-tillstånden nåbara.

När vi beräknar dessa dp-tillstånd går det såklart att spara hur v och $p[v]$ ska gå till v's olika barn för att kunna markera v's subträd. Då går A att återskapa om vi mot slutet avgör att en fullständig sekvens B existerar.

Vi kommer nu dela in noderna $a_1 = x_1, x_2, \ldots, x_q = a_N$ längs p (ordnade här i den naturliga ordningen) i fyra kategorier:

1. x_i har inga barn i det 'rotade' trädet,
2. x_i har endast löv som barn i det 'rotade' trädet,
3. x_i har exakt ett barn som inte är ett löv i det 'rotade' trädet,
4. x_i har exakt två barn som inte är löv i det 'rotade' trädet.

Notera att om en nod x_i på p har fler än två barn som inte är löv existerar det ingen fullständig sekvens B (bevis lämnas till läsaren).

Nu är vi redo för nästa dp, för sätten som vi kan markera noderna längs p, och deras barn-subträd. Följande lemma kommer att leda oss till att notera att det endast finns $O(N)$ (relevanta) dp-tillstånd:

- Betrakta den markerade noden x_χ med maximalt χ under en 'tidpunkt' i utförandet av en fullständig sekvens B. Då måste alla noder x_i längs p med $i \leqslant \chi - 2$, och alla deras barn, redan vara markerade.

Vi kommer nu lista ett antal tillstånd noderna x_i kan vara i. Denna lista är egentligen inte fullständig, men det kan visas att om det existerar en fullsätndig sekvens B så existerar det även en sådan där noderna x_i bara antar dessa tillstånd:

(a) Av typ 1, och noden är omarkerad,

(b) Av typ 1, och noden är markerad,

(c) Av typ 2, och noden är omarkerad, och barnen är omarkerade,

(d) Av typ 2, och noden är omarkerad, och alla barn är markerade,

(e) Av typ 2, och noden är markerad, och alla barn är omarkerade,

(f) Av typ 2, och noden är markerad, och alla barn är markerade,

(g) Av typ 3, och noden är omarkerad, och alla barn är omarkerade,

(h) Av typ 3, och noden är markerad, och alla barn är omarkerade,

(i) Av typ 3, och noden är markerad, och alla barn är markerade,

(j) Av typ 4, och noden är omarkerad, och alla barn är omarkerade,

(k) Av typ 4, och noden är markerad, och alla barn är markerade.

Vi kallar tillsånden (b), (f), (i) och (k) *färdiga*. Vi kommer nu beskriva alla möjliga sätt att komma till varje tillstånd för noderna. Ett dp-tillstånd är en sekvens av längd q med nod-tillståndet av varje nod x_i på p, samt vilken nod som A är vid 'nu'. Enligt lemmat ovan, och beskrivningen av nod-tillstånden ovan, följer det att det endast finns $O(N)$ dp-tillstånd.

(a) Noder av typ 1 börjar i detta tillstånd,

(b) Noden är i tillstånd (a), och hoppas till från någon annan nod på avstånd som mest 2 längs med p,

(c) Noder av typ 2 börjar i detta tillstånd,

(d) Noden x_i är i tillstånd (c), och vi gör ett drag mellan x_{i-1} och x_{i+1},

(e) Noden är i tillstånd (c), och vi gör ett drag till denna nod från en annan nod på avstånd som mest 2 längs med p,

(f) Noden är i tillstånd (c), och hoppas till från någon annan nod på avstånd 1. Eller, noden är i tillstånd (e) och vi hoppar från denna nod till en annan nod på avstånd 1. Eller, noden är i tillstånd (d) och vi hoppar till denna nod från en nod på avstånd som mest 2.

(g) Noder av typ 3 börjar i detta tillstånd.

(h) Noden är i tillstånd (g), och hoppas till från en nod på avstånd som mest 2 längs med p,

(i) Noden är i tillstånd (h), och vi går till en nod på avstånd 1. Eller, noden är i tillstånd (g) och vi hoppar till noden från en nod på avstånd 1 längs med p,

(j) Noder av typ 4 börjar i detta tillstånd.

(k) Noden är i tillstånd (j), och vi går till denna nod från en nod på avstånd 1, och går från denna nod till en nod på avstånd 1.

När vi säger att vi hoppar, eller går, till noder på p så menar vi att vi markerar dem. Alltså, de blir nästa noden i A längs med p, samtidigt som vi implicit besöker lite noder i B som inte visas, och noder i A som inte är på p.

Någon som inte nämns i denna lista—men som är imperativt—är att när det står i texten ovan att ett drag på avstånd 1 görs (eller när tillståndsförändringen till (d) sker), så kan inte någon annan tillståndsförändrig ske som kräver att draget har avstånd 1, vid den (/de) andra noden (/-erna) som hoppas till/från. Detta är eftersom dessa förändringar omfattar att det görs ett drag från en nod i subträdet av en nod x_i längs p till en av dess grannar x_{i-1} eller x_{i+1}, eller från grannarna direkt till subträdet.

I.a.f, det är klart att detta tillåter en $O(N)$ algoritm för att kolla om en fullständig sekvens B existerar. Om alla nåbara tillstånd i dp:n även sparar ett tillstånd de kan nås från, kan vi sedan återskapa B, och därmed även A, på $O(N)$. Därmed är vi färdiga. □

3 Dynamic connectivity

Problem 3. Du får givet en oriktad graf med N noder och M kanter. Det finns som mest en kant mellan varje par av noder. Varje kant är unikt numererad mellan 1 och M. För att resa i grafen måste du köpa en biljett. Varje biljett beskrivs av två heltal $1 \leqslant i \leqslant j \leqslant M$, vilket innebär att du får färdas längs med alla kanter med vars nummer k uppfyller $i \leqslant k \leqslant j$. Det finns en biljett för varje par av i, j där $1 \leqslant i \leqslant j \leqslant M$, vars kostnad är $j - i$. Du får nu Q stycken frågor. För varje fråga får du givet två noder A och B, och du ska svara med priset av den billigaste biljetten som låter dig färdas från A till B, eller avgöra att en sådan biljett ej existerar.

Om inte specifierat gäller det att $N, M \leqslant 5 \cdot 10^4$ och $Q \leqslant 1000$ för alla deluppgifter.

Deluppgift 3.1 (1 poäng). $M \leqslant 50$.

Deluppgift 3.2 (1 poäng). $M \leqslant 400$.

Deluppgift 3.3 (1 poäng). $M \leqslant 1000$.

Deluppgift 3.4 (1 poäng). $M \leqslant 5000$.

Deluppgift 3.5 (1 poäng). $M, Q \leqslant 7000$.

Deluppgift 3.6 (1 poäng). $M \leqslant 10000$.

Deluppgift 3.7 (1 poäng). $N \leqslant 1000$.

Deluppgift 3.8 (3 poäng). Inga ytterligare begränsningar.

Lösning Problem 3. Följande $O(Q(M+N)+M\min\{N,M\}\alpha(N+M))$ algoritm löser problemet för deluppgifter 1–7. Här är α inversen av Ackermannfunktionen.

Algoritmen kommer att fixera i och hitta j så att kostanden av biljett (i, j) som låter dig färdas mellan A_q och B_q är minimal, för varje fråga $1 \leqslant q \leqslant Q$.

Iterera igenom i från M till 1. Iterera sedan igenom j från i till 'M' och lägg successivt till kanten j till följande datastruktur. Varje nod håller koll på vilken sammanhängande komponent den är med i i grafen av kanterna $[i, j]$ genom union-find. När kanten j ska läggas till kollar vi först om noderna kanten förbinder redan är i samma komponent på $O(\alpha(N + M))$. Det betyder att det finns en väg av kanter $(e_1, \ldots, e_\ell)$ med $i \leqslant e_1, \ldots, e_\ell < j$ som förbinder noderna som kant j går mellan. Då kan vi slänga bort kant j för alla $i' < i$. Annars lägger vi till kant j till union-find:en på $O(\alpha(N + M))$, alltså vi merge:ar noderna som kant j förbinder. Notera att grafen $G_{i,j}$ av kanter som vi behållit i detta steget bildar en skog. Om $j = i$ svarar vi på alla frågor q med edge$[i] = (A_i, B_i)$. Om kanten j förbinder två komponenter där den ena har kant i så går vi igenom alla noder i den av de två komponenterna som inte redan innehöll i: Vi kan då svara på att det minsta j':et så att (i, j') är en biljett som det går att resa mellan A_q och B_q på är $j' = j$, för alla frågor q som har ändpunkt i en av noderna som kanten i går mellan, och andra ändpunkt i en av noderna i komponenten som nu merge:as som inte innehöll kanten i. (Hur detta har formulerats för att det funkar för $j = i$ också.)

Jag hävdar att algoritmen kör på $O(Q(M + N) + M\min\{N, M\}\alpha(N + M))$. Notera att när en kant j betraktas kommer den antingen läggas till: $G_{i,j} = G_{i,j_2} + \text{edge}[i]$, eller så kommer kanten j tas bort och aldrig betraktas mer för något $i' < i$. Varje kant slängs bort för evigt som mest 1 gång, så detta redogör för en tidskomplexitet av $O(M\alpha(N + M))$ totalt. Samtidigt kommer $G_{i,M'}$ ha som mest $\min\{N - i, M\} = O(\min\{N, M\})$ kanter tillagda i sig för varje iteration med fixerat i, vilket ger en tidskomplexitet på $O(M\min\{N, M\}\alpha(N + M))$ totalt. Varje fråga kommer att uppdateras på $O(1)$ som mest en gång per iteration för fixerat i, vilket ger en tidskomplextet på $O(MQ)$ totalt. Dessutom betraktar vi i varje iteration med fixerat i som mest $\min\{N, M + 1\}$ noder som möjligen kan ha en fråga som går mellan den noder och en av noderna som angränsar till kant i: vilket ger som mest $2\min\{N, M + 1\}$ kollar om det finns en viss fråga för varje iteration med fixerat i, eller $O(M\min\{N, M\})$ totalt. I början av programmet går vi igenom varje av de N noderna och lägger till i en hash-map alla noder som bildar ett par så att det finns en fråga som efterfrågar minsta kostanden för en biljett som kan ta sig mellan det paret noder, vilket

kan göras på $O(NQ)$. Det tillåter också att vi kollar om en nod har en fråga till de två noderna som kant i förbinder på $O(1)$.

Om två frågor har att $\{A_{q_1}, B_{q_1}\} = \{A_{q_2}, B_{q_2}\}$ kan vi merge:a dem i början. Om en fråga har $A_q = B_q$ är svaret 0 för den frågan, och den behöver inte betraktas av algoritmen.

Att slänga bort kanter j kan göras genom att spara kanterna i en länkad lista som itereras igenom för varje iteration där i är fixerat. Ovan är då j_2 indexet av kanten före j i listan (om en sådan finns, annars är G_{i,j_2} den tomma grafen på de N noderna. Notera att även om graferna G har N noder kan den återställas på $O(M)$ efter varje iteration av fixerat i. Eller så kan de återställas på $O(N)$ genom att konstruera en ny tom graf. Därmed tar det $O(\min\{N, M\})$ att konstruera den tomma grafen G i början av varje iteration med fixerat i. Samma sak med union-find:en.) Ovan är M' antalet kanter som är kvar i den länkade listan vid det relevanta tillfället. □

Ytterligare lösning Problem 3. Följande $O(QM \log N)$ algoritm löser deluppgift 8.

Påstående 3.1. *Dynamic connectivity i en oriktad graf där ordningen av delete-operationerna är känd på förhand går att lösa på $O(E \log V)$ tid, där E är antalet operationer: lägg till/ta bort en kant eller kolla om två noder är connected, och V är antalet noder, givet att $V = O(E)$.*

För alla frågor där den ena noden inte har någon kant vid sig kan vi besvara dessa separat. Om $A_q = B_q$ så kostar den billigaste biljetten 0. Annars så finns det ingen möjlig bijett för denna frågan. Hädanefter kan vi därför anta att $N = O(M)$ när vi svarar på de resterande frågorna.

Notera att om en biljett (i, j) kan ta en mellan A_q och B_q, så kommer alla biljetter (i', j') med $i' \leqslant i$ och $j \leqslant j'$ också kunna ta en mellan A_q och B_q. Låt $f_q(i)$ vara det minsta talet så att biljetten $(i, f_q(i))$ kan ta en mellan A_q och B_q. (Om det inte existerar ett sådant tal låter vi $f_q(i) = \infty$.) Notera att svaret för en fråga q kommer att vara: $\min_{1 \leqslant i \leqslant M}\{f_q(i) - i\}$ (Notera att detta är sant även då $A_q = B_q$), förutom om denna kvantitet blir ∞, vilket betyder att det inte finns någon biljett som kan ta en mellan A_q och B_q.

Från **Påstående 3.1** följer då att vi kan besvara varje fråga på $O(M \log N)$ tid—vilket ger en total tidskomplexitet av $O(QM \log N)$—på följande sätt. Betrakta den givna grafen på N noder, och förbered att ta bort kanterna i ordning av index från 1 till M. Börja med ingen kant tillagd. För varje i från 1 till M, lägg till kanten j med minsta index $j \geqslant i$ som inte redan finns i grafen, tills det går att ta sig mellan A_q och B_q (eller till alla kanter $\geqslant i$ är tillagda). Då kommer $f_q(i) = j$ (eller $f_q(i) = \infty$ om A_q och B_q inte kan nå varandra.). Sedan tar vi bort kant i (som alltid kommer att finnas) och går vidare till nästa iteration där i har ökat med ett. Notera att eftersom $f_q(i) \leqslant f_q(i+1)$, enligt disskussionen ovan, kommer detta alltid att ge rätt värden på f_q, och därmed rätt svar för fråga q.

Bevis **Påstående 3.1**. Enligt Wikipedia[1] går detta att göra med ett link/cut träd på den givna tidskomplexiteten.

Några förändringar man måste göra till den vanliga implementationen av link/cut träd är följande:

- Vi kommer att använda splay träden för att hitta kanten med minst index längs med en preferred path i link/cut trädet. Då måste varje nod i splay trädet hålla reda på kanten under och kanten över sig längs med preferred-path:en (om en sådan finns), och dess index, och minimum över alla kanter i sitt subträd i splay-trädet. Detta kan enkelt uppdateras när vi gör zig- och zag-rotationer eller merge:ar eller split:ar splay träd. Vi kommer även behöva uppdatera detta när vi reverse:ar splay-trädet, vilket inte går riktigt för edge-cases, alltså för båda noderna i början och slutet av den föredragna vägen. Det går när vi merge:ar och split:ar eftersom edge-cases noderna blir rötterna i splay-träden, vars värden KAN modifieras.

- Den enda gången vi kommer reverse:a ett splay träd är när vi vill hitta den minsta kanten på vägen mellan a och b. Först gör vi a till roten i dess link/cut träd. Detta görs genom att köra `access` på a. Sedan gör vi a till roten av link/cut trädet genom att reverse:a dess splay-träd. Detta kommer påverka tidskomplexiteten, men det går att bevisa att det som mest bildas $O(\log N)$ tunga, oföredragna (unpreferred), kanter, så tidskomplexitetsanalysen[2] blir den samma. Efter att a har blivit roten kan vi köra varianter av `access` på a, och sedan på b, som tar hand om specialfallen som händer efter reverse:andet av splayträdet. Sedan kan vi query:a den minsta kanten i den föredragna vägen från roten (a) till b (genom att query:a b's splay-träd, som vid denna tidpunkt representerar den föredragna vägen från a till b, eftersom vi just anropade `access(b)`).

Tilldela varje kant i grafen $\mathcal{G}$ en vikt lika med ordningen den kommer tas bort i. Så den första kanten som tas bort har vikt 1, den andra 2, osv. och den sista kanter har vikt som mest E. Vi vill upprätthålla en maximalt spännande skog i $\mathcal{G}$, där varje träd representeras av ett link/cut träd. Att ta bort en kant från $\mathcal{G}$ är enkelt. Om kanten inte längre finns i link/cut träden så gör vi inget. Om kanten finns så tar vi bort den. För att lägga till en kant (u, v) i $\mathcal{G}$ kollar vi först om u och v är sammanhängande. Om de inte är det kan vi anropa `link(u, v)` på u och v. Annars måste vi ta bort den minsta kanten på path:en från u till v i deras link/cut tree. Denna kan hittas (och sedan `cut`:as) enligt disskussionen ovan.

Då kan vi lägga till kanter och query:a om två noder är sammanhängande online. Varje operation tar som mest $O(\log N)$ amortized, enligt Wikipedia[3] och disskussionen ovan. □

Därmed har vi visat **Påstående 3.1** vilket betyder att algoritmen löser problemet fullständigt, med en tidskomplexitet på $O(QM \log N)$. □

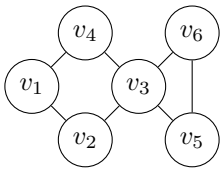
[1] `en.wikipedia.org/wiki/Dynamic_connectivity#Offline_dynamic_connectivity`

[2] `en.wikipedia.org/wiki/Link/cut_tree#O(log_2_n)_upper_bound`

[3] `en.wikipedia.org/wiki/Link/cut_tree#Improving_to_O(log_n)_upper_bound`

4 Cykler / Probabilistisk

Problem 4. Du får givet en sammanhängande, oriktad graf med N noder och M kanter. Det finns som mest en kant mellan varje par av noder. Ditt mål är att hitta en delmängd kanter som utgör en *fisk*-delgraf. Detta innebär att om du tar bort alla kanter förutom de valda, och sedan tar bort de resulterande noderna med grad 0, så är den kvarvarande komponenten isomorfisk med fiskgrafen (se Figur 1). En given graf är isomorfisk med fisk-delgrafen om man kan ändra nodernas index så att grafen blir exakt likadan som fiskgrafen.



Figur 1: Fiskgrafen

Om inte specifierat gäller det att $N \cdot M < 10^9$ för alla deluppgifter.

Deluppgift 4.1 (1 poäng). $M \leqslant 16$.

Deluppgift 4.2 (1 poäng). Grafen är en kaktus. Detta innebär att varje par av cykler (där alla noder är unika) har som mest en nod gemensamt.

Deluppgift 4.3 (1 poäng). $M \leqslant N + 10$.

Deluppgift 4.4 (7 poäng). Inga ytterligare begränsningar.

Lösning Problem 4. Följande $O(NM)$ lösning är probabilistisk. Om det finns en fiskgraf hittar lösning en sådan med sannolikhet $\geqslant 1536^{-1} = 2^{-9} \cdot 3^{-1}$ (oberoende av N och M), och om det inte finns en fiskgraf hittar lösningen inte någon sådan. Därmed, om lösningen upprepas $\frac{\log 10^{-9}}{\log(1-1536^{-1})} < 32000$ gånger, eller tills en fiskgraf hittas, så kommer lösningen ge korrekt svar med en sannolikhet på $> 1 - 10^{-9}$.

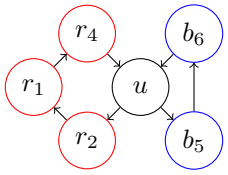
Låt G vara den givna grafen. Lösningen itererar igenom varje nod u, och gör följande på $O(M)$:

Varje nod i $\mathcal{G}\backslash\{u\}$ tilldelas slumpmässigt en av två färger: röd eller blå, samt ett (unikt) tal mellan 2 och M Låt $\mathcal{R}$ vara den inducerade subgrafen av $\mathcal{G}$ på alla röda noder, plus u, där alla kanter är riktade så att de går från en nod men lägre tal till en nod med högre tal. Om en av noderna som kanten förbinder är u väljs en riktning slumpmässigt. Definera $\mathcal{B}$ symmetriskt för de blåa noderna.

Påstående 4.1. *Om det finns distinkta blå noder $b_5, b_6 \in V(\mathcal{B})\backslash\{u\}$, och distinkta röda noder $r_1, r_2, r_4 \in \mathcal{R}\backslash\{u\}$, så att de riktade kanterna*

$$(u, b_5), (b_5, b_6), (b_6, u), (u, r_2), (r_2, r_1), (r_1, r_4), (r_4, u)$$

finns i $\mathcal{B} \cup \mathcal{R}$ så kommer följande algoritm hitta en riktad fiskgraf (se Figur 2 nedan) i $\mathcal{B} \cup \mathcal{R}$.



Figur 2: En riktad fiskgraf i $\mathcal{B} \cup \mathcal{R}$.

Från **Påstående 4.1** följer det att algoritmen kommer hitta en specifik fiskgraf med sannolikhet $2^{-5} \cdot 2^{-2} \cdot 6^{-1} \cdot 2^{-2} = 2^{-9} \cdot 3^{-1}$. Varje fiskgraf har att dess noder och kanter kan färgläggas och riktas på 2 sätt så att den blir en riktad fiskgraf. Att alla noder får rätt färg har sannolikhet 2^{-5}. Vardera av kanterna $(v_3, v_5), (v_5, v_6), (v_6, v_3)$ är uniformt slumpmässigt riktade, så sannolikheten att de bildar en 3-cykel är 2^{-2}. Vidare har kanterna $(v_3, v_2), (v_2, v_1), (v_1, v_4), (v_4, v_3)$ en sannolikhet på $6^{-1} \cdot 2^{-2}$ att bilda en 4-cykel, eftersom kanterna $(v_4, v_3), (v_3, v_2)$ riktas uniformt slumpmässigt, och sannolikheten att talen som tilldelas r_2, r_1, r_4 får kanterna $(r_2, r_1), (r_1, r_4)$ att riktas så att (u, r_2, r_1, r_4) blir en riktad 4-cykel är $(3!)^{-1} = 2^{-1} \cdot 3^{-1}$.

Nu är det dags att beskriva algoritmen och visa **Påstående 4.1**.

Beivs **Påstående 4.1**. Notera att $\mathcal{B} \backslash \{u\}$ och $\mathcal{R} \backslash \{u\}$ är DAG:er. Att det finns en riktad fiskgraf i $\mathcal{B} \cup \mathcal{R}$ är ekvivalent med att det finns en 3-cykel i $\mathcal{B}$ och en 4-cykel i $\mathcal{R}$. Dessa villkor kan kollas separat.

För att kolla om det finns en c-cykel i en graf $\mathcal{H}$ där $\mathcal{H} \backslash \{w\}$ är en DAG kan man använda följande $O(c(|E(\mathcal{H}|) + |V(\mathcal{H}|)))$ algoritm. Numerera noderna i $\mathcal{H}$ från 1 till $|V(\mathcal{H})|$. Ha ett bitset där bit:en för w är på och alla andra är av. I c iterationer; skapa ett nytt bitset genom att iterera igenom noderna i $\mathcal{H}$ och slå på alla noder som kan nås av en nod som är på i förra iterationens bitset. Om det inte är sista iterationen, slå av biten för w i det nya bitsetet. Om det efter c iterationer är så att biten för w är på så finns det en c-cykel i $\mathcal{H}$. Varje nod kan för varje iteration där den var på i bitsetet spara vilken nod som slog på den, så kan c-cykeln enkelt återkonstrueras.

Då kan vi hitta om det finns en 3-cykel i $\mathcal{B}$ och en 4-cykel i $\mathcal{R}$ på tid: $3(|E(\mathcal{B})| + |V(\mathcal{B})|) + 4(|E(\mathcal{R})| + |V(\mathcal{R})|) \leqslant 7N + 7M = O(M)$, och konstruera dem om de finns. Därmed vi även hitta om det finns, och konstruera i så fall, en riktad fiskgraf i $\mathcal{B} \cup \mathcal{R}$ på $O(M)$. □

Därmed är **Påsteånde 4.1** bevisat. Därmed kommer vår algoritm för varje iterationsomgång genom alla u:n hitta en riktad fiskgraf, om det finns någon fiskgraf i $\mathcal{G}$, med sannolikhet $\geqslant 2^{-9} \cdot 3^{-1} = 1536^{-1}$. Därmed kommer < 32000 iterationer av algoritmen att hitta en fiskgraf, om det finns någon, med sannolikhet $> 1 - 10^{-9}$ på $O(NM)$.

Detta löser problemet fullständigt. □