

2-sammenhengende grafer og sterkt sammenhengende komponenter

Denne uken fokuserer vi på avanserte egenskaper i grafer, spesielt 2-sammenheng i urettede grafer og sterkt sammenhengende komponenter i rettede grafer.

Oversikt Vi skal lære: 1. 2-sammenhengende urettede grafer ○ Hva det betyr at en graf er 2-sammenhengende. ○ Hvordan sjekke effektivt om en graf er 2-sammenhengende. 2. Sterkt sammenhengende komponenter i rettede grafer ○ Hva det betyr at en komponent er sterkt sammenhengende. ○ Hvordan finne alle sterkt sammenhengende komponenter effektivt.

Hva er en 2-sammenhengende graf? • En urettet graf er 2-sammenhengende dersom: ○ Grafen er sammenhengende, og ○ Fjerning av hvilken som helst enkelt node ikke bryter sammenheng. • Med andre ord: grafen har ingen artikulasjonspunkter. • Viktig egenskap: ○ Hvis en graf er 2-sammenhengende, finnes det to uavhengige stier mellom alle par av noder. ○ Dette er relevant i nettverksdesign, hvor man ønsker feiltoleranse.

Hvordan sjekke 2-sammenheng • Bruk et dybde-først søk (DFS) med følgende ideer: 1. Registrer for hver node u : § $dfs_num[u]$ – tidspunktet u ble besøkt i DFS. § $low[u]$ – den laveste dfs_num som kan nås fra u gjennom barn og tilbakekanter. 2. En node u er et artikulasjonspunkt hvis den har et barn v slik at $low[v] \geq dfs_num[u]$. 3. Hvis ingen artikulasjonspunkter finnes, er grafen 2-sammenhengende. • Tidskompleksitet: $O(|V| + |E|)$ med DFS.

Sterkt sammenhengende komponenter (SCC) • En rettet graf er sterkt sammenhengende dersom: ○ For alle noder u og v , finnes det en sti fra $u \rightarrow v$ og $v \rightarrow u$. • En sterkt sammenhengende komponent (SCC) er et maksimalt sett av noder som er sterkt sammenhengende. ○ Maksimalt betyr at ingen andre noder kan legges til uten å bryte sterk sammenheng.

Hvordan finne SCC effektivt • Kosaraju's algoritme: 1. Kjør DFS og noter ferdigstillelsestid for hver node. 2. Transponer grafen (bytt retning på alle kanter). 3. Kjør DFS igjen, i rekkefølge av synkende ferdigstillelsestid. 4. Hver DFS-traversering finner en SCC. • Tar $O(|V| + |E|)$ tid. • Tarjan's algoritme: ○ Bruker én DFS og en stack for å finne SCC. ○ Holder styr på low-link verdier (ligner på 2-sammenhengsjekk). ○ Også $O(|V| + |E|)$ tid.

2-sammenhengende grafer

Sammenhengende grafer • En urettet graf er sammenhengende hvis det finnes en sti mellom alle par av noder. • Hvis grafen ikke er sammenhengende, kan den deles inn i komponenter, der hver komponent er en maksimal sammenhengende delgraf. • For å sjekke om en graf er sammenhengende: ○ Bruk et dybde-først søk (DFS). ○ Start DFS fra en vilkårlig node. ○ Hvis DFS besøker alle noder, er grafen sammenhengende. • Tidskompleksitet: $O(|V| + |E|)$. • Viktig: dette gir oss grunnlaget for å definere 2-sammenheng, fordi det forutsetter at grafen allerede er sammenhengende.

2-sammenheng • En graf $G=(V,E)$ er 2-sammenhengende dersom: ○ Grafen forblir sammenhengende selv om hvilken som helst enkelt node $v \in V$ fjernes. • Alternativ definisjon: ○ Mellom alle par noder u og v finnes to distinkte stier. ○ To stier er distinkte dersom de ikke deler noen noder eller kanter, bortsett fra start- og sluttnoden u og v . • Generell definisjon: ○ En graf er k -sammenhengende dersom den forblir

sammenhengende når man fjerner mindre enn k noder. • Praktisk betydning: ○ Gir redundans og feiltoleranse i nettverk og infrastruktur. ○ Eksempel: § I et datanettverk: selv om en maskin går ned, vil data fortsatt nå alle andre maskiner. § I kollektivtransport: hvis én holdeplass stenges, finnes alternative ruter slik at alle reisende fortsatt kan komme frem.

Er grafen 2-sammenhengende? (naiv metode)

Naiv algoritme • Utgangspunkt: definisjonen av 2-sammenheng. • En graf er 2-sammenhengende hvis den forblir sammenhengende når en hvilken som helst node fjernes. • Dette gir en enkel (men ineffektiv) algoritme: Pseudokode: ALGORITHM: IsBiconnectedNaive(G) Input: En sammenhengende graf $G = (V, E)$ Output: true hvis grafen er 2-sammenhengende, false ellers 1 Procedure IsBiconnectedNaive(G) 2 for hver node $v \in V$ do 3 $G' \leftarrow$ grafen G med noden v fjernet 4 $visited \leftarrow$ empty set 5 $u \leftarrow$ en vilkårlig node $u \in G'$ 6 DFSVisit($G', u, visited$) 7 if $visited \neq V'$ then 8 return false 9 return true • Forklaring: 1. Fjern én node om gangen. 2. Kjør DFS fra en gjenværende node. 3. Hvis alle gjenværende noder ikke er besøkt, er grafen ikke 2-sammenhengende. 4. Hvis alle noder testes uten å finne brudd, er grafen 2-sammenhengende. • Tidskompleksitet: ○ For hver node: DFS tar $O(|V| + |E|)$. ○ Totalt: $O(|V| \cdot (|V| + |E|))$, som kan forenkles til $O(|V|^3)$ for tett graf.

Separasjonsnoder (articulation points) • En separasjonsnode er en node som holder grafen sammenhengende. • Hvis en separasjonsnode fjernes, splittes grafen i flere komponenter. • Formelt: hvis alle stier mellom to noder går gjennom samme node $v \in V$, er v en separasjonsnode. • Viktighet: ○ Hvis grafen har ingen separasjonsnoder, er den 2-sammenhengende. • Det finnes en effektiv algoritme som finner alle separasjonsnoder i $O(|V| + |E|)$. ○ Dermed kan man sjekke 2-sammenheng på en effektiv måte uten å teste hver node manuelt.

Separasjonsnoder (articulation points) via DFS Intuisjon • En separasjonsnode er en node som holder grafen sammenhengende. • Hvis vi fjerner den, splittes grafen i flere komponenter. • Vi kan finne separasjonsnoder effektivt ved å bruke et dybde-først søk (DFS).

DFS-spenntre • Kjør DFS på grafen G , og bygg et spenntre T : ○ Dersom vi går fra u til v i søket, legger vi til en discovery-edge (u, v). ○ I tillegg holder vi styr på back-edges: kanter som peker til allerede oppdagede noder. • Sentrale punkter: 1. Roten r i DFS-treet: Hvis roten har mer enn ett barn, er roten en separasjonsnode. 2. Indre noder u : Hvis u har et barn v der ingen etterkommere av v (inkludert v selv) har en back-edge til en forgjenger av u , er u en separasjonsnode.

Eksempellustrasjon DFS-tre (discovery edges): A /

B C / \

D E F / G Back-edges:

- Kan gå fra E til A
- Kan gå fra F til B • Roten A har 2 barn \rightarrow A er separasjonsnode. • Node B har et barn D, men D kan nå A via back-edge? Hvis ikke \rightarrow B er separasjonsnode. • Node C har ett barn F, men F kan nå en forgjenger av C via back-edge \rightarrow C er kanskje ikke separasjonsnode.

Depth og Low • For å finne separasjonsnoder systematisk: ○ $depth[u]$: dybden til node u i DFS-treet. ○ $low[u]$: den laveste dybden som kan nås fra u via: § Etterkommere av u § Maksimalt én back-edge • Regel for separasjonsnode: ○ Hvis v er et barn av u i DFS-treet, og $depth[u] \leq low[v]$, så er u en separasjonsnode. ○ Intuisjon: v (og dens etterkommere) kan ikke nå noen forgjengere av u uten å gå gjennom u .

Oppsummering av algoritmen 1. Kjør DFS på grafen og noter $\text{depth}[u]$ for hver node. 2. Beregn $\text{low}[u]$ for hver node u under DFS. 3. Marker node u som separasjonsnode dersom: \bigcirc u er roten med mer enn ett barn, eller \bigcirc For noe barn v av u : $\text{depth}[u] \leq \text{low}[v]$ 4. Algoritmen kjører i $O(|V| + |E|)$.

Finne separasjonsnoder i en sammenhengende graf Formål • Å identifisere alle noder i grafen G som holder grafen sammenhengende. • Fjerner man en separasjonsnode, vil grafen splittes i flere komponenter. • Effektiv algoritme bruker DFS med depth og low .

Hovedidé 1. Kjør DFS fra en vilkårlig startnode s . 2. For hver node u holder vi styr på: \bigcirc $\text{depth}[u]$: dybden til u i DFS-treet. \bigcirc $\text{low}[u]$: den laveste dybden som kan nås fra u via en etterkommer eller én back-edge. 3. En node u er en separasjonsnode hvis: \bigcirc u er roten og har mer enn ett barn, eller \bigcirc u ikke er roten, og det finnes et barn v slik at $\text{low}[v] \geq \text{depth}[u]$.

Pseudokode forklaring Prosedyre SeparationVertices(G): • Initialiserer maps for depth og low , samt et sett seps for separasjonsnoder. • Velger en vilkårlig startnode s , setter $\text{depth}[s] = 0$ og $\text{low}[s] = 0$. • Teller antall barn i DFS-treet som children. • For hver nabo u av s som ikke allerede er besøkt, kaller den rekursiv funksjon SepRec($G, s, u, 1$). • Hvis roten har mer enn ett barn \rightarrow legg til i separasjonsnoder. Prosedyre SepRec(G, p, u, d) (rekursiv): • Setter $\text{depth}[u] = d$ og $\text{low}[u] = d$. • For hver nabo v av u : 1. Hvis v er forelderen $p \rightarrow$ hopp over. 2. Hvis v allerede er besøkt \rightarrow oppdater $\text{low}[u] = \min(\text{low}[u], \text{depth}[v])$. 3. Hvis v ikke er besøkt \rightarrow kall SepRec($G, u, v, d+1$) rekursivt. § Oppdater $\text{low}[u] = \min(\text{low}[u], \text{low}[v])$. § Hvis $d \leq \text{low}[v] \rightarrow u$ er separasjonsnode \rightarrow legg til seps.

Intuisjon bak low og depth • $\text{depth}[u]$: hvor dypt node u er i DFS-treet. • $\text{low}[u]$: hvor «tidlig» (lav dybde) kan man komme fra node u eller dens etterkommere ved å bruke maks én back-edge. • Hvis $\text{low}[v] \geq \text{depth}[u] \rightarrow$ ingen etterkommere av v kan nå forgjengere av u uten å gå gjennom $u \rightarrow u$ er kritisk.

Kjøretidsanalyse • DFS besøker hver node én gang $\rightarrow O(|V|)$ • Hver kant behandles maksimalt to ganger (discovery/back-edge) $\rightarrow O(|E|)$ • Totalt: $O(|V| + |E|)$

Oppsummering • Algoritmen gir oss alle separasjonsnoder effektivt. • Kan brukes til å sjekke om en graf er 2-sammenhengende: \bigcirc Grafen er 2-sammenhengende hvis og bare hvis ingen separasjonsnoder finnes. • Denne metoden er mye raskere enn den naive løsningen som fjerner hver node og sjekker grafens sammenheng.

Sjekke om en graf er 2-sammenhengende Formål • Å avgjøre om grafen forblir sammenhengende selv om hvilken som helst node fjernes. • Grafen G er 2-sammenhengende hvis og bare hvis den har ingen separasjonsnoder.

Hovedidé 1. Finn alle separasjonsnoder i grafen ved hjelp av DFS-algoritmen SeparationVertices(G). 2. Dersom settet av separasjonsnoder er tomt \rightarrow grafen er 2-sammenhengende. 3. Dersom det finnes minst én separasjonsnode \rightarrow grafen er ikke 2-sammenhengende.

Pseudokode ALGORITHM: ER GRAFEN 2-SAMMENHENGENDE? Input: En sammenhengende graf $G = (V, E)$ Output: true hvis G er 2-sammenhengende, false ellers 1 Procedure IsBiconnected(G) 2 return SeparationVertices(G) is empty • SeparationVertices(G) er funksjonen som finner alle separasjonsnoder i grafen. • Metoden er effektiv: den kjører i $O(|V| + |E|)$ tid, som er mye bedre enn den naive løsningen som fjerner hver node og kjører DFS på nytt.

Intuisjon • Hvis grafen ikke har noen separasjonsnoder, kan ingen enkeltnode splitte grafen \rightarrow derfor er grafen robust mot fjerning av én node. • Hvis grafen har minst én separasjonsnode, finnes det minst ett

kritisk punkt som, når fjernes, deler grafen → ikke 2-sammenhengende.

Bruk i praksis • Denne sjekken er nyttig i nettverksdesign, der man ønsker redundans. • For eksempel: ○ Datamaskinnettverk → pakker skal kunne rutes via alternative stier selv om én maskin går ned. ○ Transportnettverk → det bør finnes alternative ruter hvis en holdeplass stenges.

Sterkt sammenhengende komponenter (SCC) Definisjon • En rettet graf er sterkt sammenhengende dersom det finnes en sti mellom alle par av noder i grafen. • En sterkt sammenhengende komponent (SCC) er en delgraf av grafen som oppfyller: 1. Det finnes en sti mellom alle par noder i komponenten. 2. Komponenten er maksimal, dvs. ingen flere noder kan legges til uten at egenskapen brytes. Intuisjon: Tenk på SCC som «sykler» eller tett sammenkoblede grupper i grafen.

Eksempel • Grafen kan ha flere SCC-er: ○ En komponent med noder {A, B, C, D} hvor alle kan nå hverandre via stier. ○ En annen komponent med noder {E, F, G, H}.

Reversert graf • Den reverserte grafen til G , kalt G_r , er grafen der alle kantene er snudd: ○ Hvis $(u,v) \in E \rightarrow (v,u) \in E_r$ • Viktig egenskap: G og G_r har alltid de samme SCC-ene. Dette brukes i algoritmer som Kosaraju for effektivt å finne alle SCC-er i en graf.

Praktisk betydning • SCC-er brukes til å forstå strukturen i et nettverk: ○ Programavhengigheter i software ○ Nettverk med retninger, f.eks. trafikkflyt, web-sider ○ Optimalisering og komponentanalyse

Reverser graf Definisjon • Den reverserte grafen G_r av en graf $G=(V,E)$ er en graf med de samme nodene, men med alle kantene vendt i motsatt retning. ○ Hvis $(u,v) \in E \rightarrow (v,u) \in E_r$

Algoritme for å reversere en graf Input: En graf $G=(V,E)$ Output: Den reverserte grafen $G_r=(V,E_r)$

Procedure ReverseGraph(G) $E_r \leftarrow$ empty set for $(u, v) \in E$ do add (v, u) to E_r return (V, E_r) Forklaring: • Vi går gjennom alle kantene i grafen og snur dem. • Dette er en enkel operasjon som tar $O(|E|)$ tid.

Finne sterkt sammenhengende komponenter (SCC) Intuisjon • For en node v , SCC-en som inneholder v består av alle noder som kan nås fra v og som også kan nå v . • Metode: 1. Finn alle noder som kan nås fra v i grafen G . 2. Finn alle noder som kan nås fra v i den reverserte grafen G_r . 3. Snittet av disse mengdene er SCC-en til v . Kjøretid • Denne metoden tar $O(|V| \cdot (|V| + |E|))$ hvis man gjør et søk for hver node. • Med mer sofistikerte algoritmer som Kosaraju eller Tarjan kan vi redusere dette til $O(|V| + |E|)$.

Praktisk bruk • SCC-analyse hjelper med å finne «sykliske» deler av grafen: ○ Avhengighetsanalyse (software, pakker) ○ Nettverksanalyse (trafikk, kommunikasjon) ○ Optimalisering (komponenter som alltid må behandles sammen)

Komponentgrafen Definisjon • Etter at vi har identifisert alle sterkt sammenhengende komponenter (SCC-er) i en rettet graf G , kan vi anslå hver SCC som én node. • Dette gir oss en ny graf kalt komponentgrafen. Egenskaper • Asyklisk: Komponentgrafen inneholder ingen sykler, selv om den opprinnelige grafen G hadde sykler. • SCC som noder: Hver node i komponentgrafen representerer en hel SCC i G . • Noder i toppen: Noder i den topologisk siste komponenten kan ikke nå noen andre komponenter. Dette er nyttig for analyser av avhengigheter eller flyt i nettverk.

Illustrasjon • Tenk deg en graf G med SCC-er: ○ $SCC_1=\{A,B,C\}$ ○ $SCC_2=\{D,E\}$ ○ $SCC_3=\{F,G,H\}$ •

Komponentgrafen vil representere dette som tre noder, med kanter mellom dem hvis det finnes kanter mellom SCC-ene i G .

Topologisk ordning av komponentgrafen Definisjon • En graf har en topologisk ordning dersom vi kan liste nodene slik at for hver kant (u,v) , kommer u før v . • Dette er bare mulig hvis grafen er asyklisk, derfor kan vi topologisk sortere komponentgrafen. Sammenheng med DFS • DFSTopSort gir nodene i en rekkefølge som respekterer komponentgrafens struktur: ○ Den siste noden som blir ferdigbehandlet i DFS er topologisk første i komponentgrafen. ○ Den topologisk første noden i DFS på den reverserte grafen blir den topologisk siste i komponentgrafen. Bruk • Ved å utforske noder i motsatt topologisk rekkefølge, garanterer vi at vi ikke krysser grenser mellom SCC-er. • Dette er essensielt for algoritmer som Kosaraju for å finne alle SCC-er effektivt i $O(|V| + |E|)$ tid.

Kosaraju-algoritmen – Oversikt Kosaraju-algoritmen brukes for å finne sterkt sammenhengende komponenter (SCC) i en rettet graf G . Den utnytter egenskapene til DFS og den reverserte grafen. Trinnene i korte trekk: 1. Første DFS på grafen G ○ Utfør et fullstendig dybde-først søk. ○ Når en node er ferdigbehandlet (alle naboer besøkt), legg den på en stack. ○ Dette gir en topologisk rekkefølge av nodene i grafens komponentgraf. 2. Reverser grafen ○ Lag den reverserte grafen G_r hvor alle kanter snus. ○ Hvis $(u,v) \in E$, så blir $(v,u) \in E_r$. 3. Andre DFS på G_r ○ Utfør et nytt fullstendig DFS, men utforsk nodene i stack-rekkefølgen fra første DFS. ○ Alle noder som nås i ett DFS-kall utgjør en SCC. ○ Gjenta til alle noder er besøkt. Hvorfor fungerer det? • Stacken sørger for at vi alltid starter DFS i G_r fra den topologisk siste komponenten i komponentgrafen. • Dette garanterer at DFS i G_r ikke krysser grenser mellom SCC-er, og at hver DFS finner en hel komponent.

Kosaraju – Pseudokode Input: En rettet graf $G=(V,E)$ Output: Alle sterkt sammenhengende komponenter i G Procedure StronglyConnectedComponents(G) $stack \leftarrow DFSTopSort(G)$ # DFS på G , legg ferdigbehandlede noder på stack $Gr \leftarrow ReverseGraph(G)$ # Lag den reverserte grafen $visited \leftarrow \text{empty set}$ $components \leftarrow \text{empty set}$ while $stack$ is not empty do $u \leftarrow stack.pop()$ if u not in $visited$ then $component \leftarrow \text{empty set}$ DFSVisit($Gr, u, visited, component$) # DFS fra u i Gr add $component$ to $components$ return $components$ Notater • DFSVisit markerer alle noder som besøkt og samler dem i én SCC. • Algoritmen har kjøretid $O(|V| + |E|)$, fordi hver kant og node besøkes maksimalt én gang i hvert DFS-kall. • Kosaraju er enkel å implementere og svært effektiv for store grafer.