Grafer – Introduksjon

```
• Hovedidé:
    ○ Grafer er datastrukturer som representerer hvordan ting henger
sammen.
    ○ De består av noder (vertices) og kanter (edges) som forbinder nodene.
• Eksempler på anvendelser:
    ○ Veinettverk og ruting
    ○ Sosiale nettverk
    ○ Oppgaveavhengigheter (topologisk sortering)
    ○ Elektriske kretser
    ○ Nettverksflyt og trafikkoptimalisering
• Ulike typer grafer:
    ○ Rettet vs. urettet: Kantene har retning eller ikke
    ○ Vektede vs. ikke-vektede: Kantene har kostnad/vekt eller ikke
    ○ Sammenhengende vs. usammenhengende: Det finnes vei mellom alle par av
noder eller ikke
```

Representasjon av grafer • Adjacency list (naboliste): ○ Hver node peker til en liste av noder den er koblet til. ○ Effektivt for sparsomt koblede grafer (få kanter). • Adjacency matrix (naboliste-matrise): ○ En NxN matrise A der A[i][j]=1 hvis det finnes kant fra i til j, ellers 0. ○ Effektivt for tett koblede grafer. • Fordeler og ulemper: ○ Liste: sparsom lagring, iterasjon over naboer rask ○ Matriser: enkel tilgang for kant-eksistens, men mye plass for sparsom graf

Traversering av grafer • Hovedformål: Besøke alle noder på en strukturert måte. • To klassiske metoder: 1. DFS – Depth-First Search (dybde-først) § Utforsker så langt som mulig langs en gren før tilbakelegging § Bruker gjerne rekursjon eller stack § Kan brukes til å oppdage sykluser, finne komponenter, og topologisk sortering 2. BFS – Breadth-First Search (bredde-først) § Utforsker alle naboer først før neste nivå § Bruker kø (queue) § Brukes for korteste vei i uvektede grafer • Eksempel på bruk: ○ BFS finner korteste vei i antall kanter ○ DFS brukes til å finne sterk sammenhengende komponenter

Topologisk sortering • Hovedidé: ○ Gjelder kun rettede acykliske grafer (DAG) ○ Vi ønsker å ordne nodene slik at for hver kant u→v, kommer u før v • Brukseksempler: ○ Planlegging av oppgaver med avhengigheter ○ Bygging av programmoduler der noen moduler krever andre først ○ Prioritering av prosesser i operativsystem • Metoder: 1. DFS-baserte: § Utfør DFS § Når en node ferdigbehandles, legg den foran i sorteringsliste § Sikkerstiller at alle avhengigheter er på plass før noden 2. Kahn’s algoritme (BFS-basert): § Finn noder med indegree 0 (ingen innkommende kanter) § Legg dem til i sorteringsrekkefølge § Fjern nodene og kantene fra grafen § Gjenta til grafen er tom • Kjøretid: ○ Begge metoder går i O(V+E), der V er antall noder og E antall kanter

Plan for de neste forelesningene • Neste forelesning – grafer med vekter: ○ Korteste stier (Dijkstra, Bellman-Ford) ○ Minimale spenntrær (Prim, Kruskal) • Forelesningen etter – komponentanalyse: ○ 2-sammenhengende grafer og separasjonsnoder ○ Sterkt sammenhengende komponenter

Eksempler på grafer Grafer brukes til å modellere mange forskjellige typer relasjoner og strukturer: • Kart: ○ Veinettverk, metro-linjer, flyruter. ○ Nodene representerer steder, kantene representerer forbindelser

mellom steder. • Bekjentskapsgrafer: ○ Sosiale nettverk der personer er noder og vennskap/kontakter er kanter. • Nettverk: ○ Datakommunikasjonsnettverk, strømnett, internett. • Tilstander: ○ Tilstandsdiagrammer i programmering og automater, hvor noder er tilstander og kanter representerer overganger. • Trær: ○ Spesialtilfelle av grafer uten sykluser, ofte brukt for hierarkier, katalogsystemer eller parsing av uttrykk.

Formell definisjon av en graf En graf G består av to mengder: • V – mengde av noder (vertices) • E – mengde av kanter (edges), der hver kant forbinder to noder fra V Eksempel: • Noder: {A,B,C,D,E,F,G} • Kanter: {{A,B},{A,C},{A,D},{B,C},{C,D},{C,F},{D,E},{E,F},{F,G}} Da har vi en graf G=(V,E). Merk: I en urettet graf er kanten {u,v} den samme som {v,u}. I en rettet graf (digraf) representeres kanten ofte som (u,v), som betyr at det går en kant fra u til v.

Terminologi Betegnelse Forklaring Eksempel / Notat Parallelle kanter Flere enn én kant mellom to noder To veier mellom A og B (Enkle) løkker En kant som starter og slutter på samme node Kant fra A til A Urettet / rettet Kantene har ingen retning vs. har retning U: veinett, R: flyruter Vektet / uvektet Kantene har en verdi (kostnad, avstand, tid) eller ikke Vektet: avstand på vei, uvektet: vennskap Enkel graf Graf uten løkker, parallelle kanter, retning eller vekt Standard sosialgraf

Ekstra notater • Grad (degree) til en node i urettet graf: antall kanter som går til/fra noden. • Indegree / outdegree i rettet graf: antall innkommende og utgående kanter. • Kompleksitet: Grafer kan være sparsomme (få kanter, $E \ll V^2$) eller tette (mange kanter, $E \approx V^2$).

Grafer med og uten retning • En graf kan ha rettede kanter (eng: directed edges) eller urettede kanter (undirected edges). • Urettede grafer: ○ Kantene har ingen retning. ○ Representasjon: {u,v}, som betyr at det går en kant mellom nodene u og v. ○ Kanten kan «krysses» begge veier: fra u til v og fra v til u. ○ Eksempel: veinett der trafikk kan gå begge veier. • Rettede grafer: ○ Kantene har retning. ○ Representasjon: (u,v), som betyr at det går en kant fra u til v. ○ Kanten kan kun følges i den angitte retningen. ○ Eksempel: flyruter fra en by til en annen, én vei. • Konvertering: ○ En urettet graf kan representeres som en rettet graf ved å legge til to kanter for hver urettet kant: (u,v) og (v,u). ○ Dette gjør det enklere å bruke algoritmer som er designet for rettede grafer på urettede grafstrukturer.

Veier og stier • Sti (path): ○ En sekvens av noder (v0,v1,...,vk) hvor hver påfølgende nodepar (vi,vi+1) er forbundet med en kant. ○ Ingen node gjentas i stien. ○ Eksempel: A→B→C→D • Vei (walk): ○ En sekvens av noder hvor hver påfølgende nodepar er forbundet med en kant, men kanter kan gjentas. ○ Noder kan også gjentas. ○ Eksempel: A→B→C→A→D • Korteste stier: ○ Ofte ønsker vi å finne stien med minst antall kanter eller minst kostnad mellom to noder. ○ Dette blir hovedtema i neste forelesning når vi ser på grafer med vekter.

Sammenhengende grafer og komponenter • En sammenhengende graf er en graf der det finnes en sti mellom alle par av noder. ○ Dette betyr at man kan gå fra hvilken som helst node til hvilken som helst annen node via kantene i grafen. ○ Eksempel: et veinett hvor alle byer er tilgjengelige fra alle andre byer. • En graf som ikke er sammenhengende består av flere komponenter: ○ En komponent er en delgraf som er sammenhengende i seg selv, men ikke har kanter til andre komponenter. ○ Man kan tenke på hver komponent som en isolert "øy" av noder som henger sammen. • I rettede grafer introduseres begrepet sterkt sammenhengende komponenter (strongly connected components, SCC): ○ En SCC er en delgraf der det finnes en sti i begge retninger mellom alle noder. ○ Dette er viktig når man analyserer systemer med avhengigheter eller prosesser som må kunne nå hverandre i begge retninger. ○ Temaet dekkes mer detaljert i senere forelesninger.

Sykler • En sykel er en sti som starter og slutter på samme node. ○ For urettede grafer kreves minst tre noder i stien for at det skal regnes som en sykel. ○ For rettede grafer må kantene i syklene følge riktig retning; her kan syklusen bestå av færre noder. • En graf som ikke inneholder noen sykluser kalles asyklisk. • Viktige spesialtilfeller av asykliske grafer: ○ Urettet, sammenhengende og asyklisk graf: kalles et tre. § Trær brukes mye i datastrukturer, hierarkier og søk. ○ Rettet og asyklisk graf: kalles en DAG (Directed Acyclic Graph). § DAG-er er svært anvendelige for å representere avhengigheter, som oppgaver i et prosjekt, beregningsgrafer i maskinlæring, eller pakkehåndtering i programmering. • DAG-er har egenskaper som gjør dem spesielt nyttige: ○ De kan topologisk sorteres (ordre som respekterer alle retninger). ○ Det finnes ingen uendelige rekursive løkker fordi de er asykliske.

Graden til en node • Graden til en node deg(v) beskriver hvor mange kanter som kobler noden v til andre noder. ○ Eksempel: I en urettet graf har en node grad 5 hvis den har fem kanter til fem andre noder. ○ Noder med kun én kant har grad 1. • Rettede grafer har to typer grad: 1. In-grad: antall kanter som går inn til noden. 2. Ut-grad: antall kanter som går ut fra noden. ○ Eksempel: I en rettet graf kan en node ha inngrad 2 og utgrad 3. ○ Resten av nodene kan ha inngrad 1 eller utgrad 1, avhengig av koblingene. • Grad gir viktig informasjon om hvor sentral en node er i grafen, og brukes i algoritmer som traversering, topologisk sortering, og sirkulæritetskontroll.

Størrelse av grafer • I en enkel komplett graf er alle noder koblet til alle andre noder. ○ Antall kanter i en slik graf med $|V|$ noder: $|E|=|V|(|V|-1)2$ ○ Dette er fordi hver node kan kobles til $|V|-1$ andre, men vi teller hver kant bare én gang. • Når vi gjør asymptotisk analyse: $O(|V|(|V|-1)2)=O(|V|2)$ ○ Dette betyr at algoritmer på komplette grafer kan få kvadratisk kompleksitet i antall noder. • Kjøretidskompleksitet for grafalgoritmer avhenger både av: ○ $|V|$ – antall noder ○ $|E|$ – antall kanter • Typer grafer etter tetthet: ○ Tette grafer (dense): mange kanter relativt til antall noder. Eksempel: sosiale nettverk med mange forbindelser. ○ Tynne grafer (sparse): få kanter relativt til antall noder. Eksempel: veinett på landet, der få byer er direkte koblet. • Viktig å merke: antall kanter er begrenset av antall noder, men ikke omvendt. ○ En graf kan ha få noder og veldig mange kanter (tett), eller mange noder og få kanter (tynn).

Representasjon av grafer Når vi jobber med grafer i algoritmer og datastrukturer, må vi velge en måte å representere grafen på. De to mest brukte metodene er: 1. Nabomatrise (Adjacency Matrix) 2. Naboliste (Adjacency List) Målet med begge er å kunne: • Sjekke om det finnes en kant mellom to noder u og v • Finne alle naboene til en node effektivt Vi antar at vi har tilgang på raske oppslag, f.eks. gjennom HashMap/HashSet i Java eller dict/set i Python, som gir konstant tid for enkelte operasjoner.

Nabomatrise • En matrise A av størrelse $|V|\times|V|$, der hver rad og kolonne representerer en node. • Verdien A[u][v]=1 hvis det finnes en kant mellom u og v, ellers 0. Eksempel: A B C D E F G A 0 1 1 1 0 0 0 B 1 0 1 0 0 0 0 C 1 1 0 1 0 1 0 D 1 0 1 0 1 0 0 E 0 0 0 1 0 1 0 F 0 0 1 0 1 0 1 G 0 0 0 0 0 1 0 Egenskaper: • Passer best for tette grafer der mange noder er koblet sammen. • Minnebruk: $O(|V|2)$ – uavhengig av antall kanter. • Sjekke naboer: Konstant tid O(1) for å sjekke om u og v er koblet. • Finne alle naboer: Krever $O(|V|)$ tid, noe som kan være ineffektivt i tynne grafer.

Naboliste • For hver node v lagres en liste eller mengde med alle naboene. Eksempel: A : [B, C, D] B : [A, C] C : [A, B, D, F] D : [A, C, E] E : [D, F] F : [C, E, G] G : [F] Egenskaper: • Passer best for tynne grafer, der få noder er koblet sammen. • Minnebruk: $O(|V|+|E|)$ – mer effektivt for sparse grafer. • Finne alle naboer til en node v: Krever O(deg(v)), som ofte er raskere enn $O(|V|)$. • Kan implementeres med ordbok + liste eller ordbok + mengde hvis vi ønsker konstant tid for oppslag mellom to noder.

Sammenligning Nabomatrise vs Naboliste Egenskap Nabomatrise Naboliste Minnebruk (O(V Sjekke kant mellom to noder O(1) O(deg(v)) eller O(1) med mengde Finne alle naboer (O(V Egnet for Tette grafer

Tynne grafer

Graftraversering Å traversere en graf betyr å gå gjennom alle nodene og kantene på en systematisk måte, ofte med et spesifikt formål: • Finne hvilke noder som kan nås fra en gitt startnode s. • Bestemme antall komponenter i grafen. • Oppdage sykluser eller spesifikke egenskaper i grafen. To vanlige strategier for graftraversering er: 1. Dybde-først søk (DFS – Depth-First Search) 2. Bredde-først søk (BFS – Breadth-First Search)

Dybde-først søk (DFS) Intuisjon: • DFS følger alltid en sti så langt den kan gå fra startnoden, før den backtracker. • Den går "dypt" i grafen før den utforsker alternative veier. Trinnvis: 1. Velg en startnode s. 2. Gå til en ubesøkt nabo. 3. Gjenta steg 2 fra denne nye noden til du ikke finner flere ubesøkte naboer. 4. Backtrack til forrige node som har ubesøkte naboer, og fortsett. 5. Avslutt når alle noder som kan nås fra startnoden er besøkt. Egenskaper: • DFS bruker rekursjon eller en stack for å holde styr på nodene som skal besøkes. • Hver node legges på stacken én gang. • Typisk minnebruk er lavere enn BFS, spesielt i tette grafer, fordi du bare trenger å lagre én sti av gangen. • DFS kan brukes til: ○ Oppdage sykluser i grafen ○ Finne sammenhengende komponenter ○ Topologisk sortering i DAGer Visualisering: Tenk på DFS som å følge en sti så langt du kan, og så spore tilbake når du støter på en blindvei. Implementasjon (rekursiv): def DFS(graph, node, visited): visited.add(node) for neighbor in graph[node]: if neighbor not in visited: DFS(graph, neighbor, visited) Alternativ med eksplisitt stack: def DFS_stack(graph, start): visited = set() stack = [start] while stack: node = stack.pop() if node not in visited: visited.add(node) stack.extend(graph[node]) Oppsummering: • DFS er enkel, effektiv og fleksibel. • Egner seg godt når vi ønsker å utforske hele grafen eller finne dybdebaserte egenskaper.

Dybde-først søk (DFS) DFS er en metode for å traversere en graf ved å følge stier dypt først før man backtracker. Den kan implementeres rekursivt eller iterativt med en stack.

1. Rekursivt DFS Idé: • Start fra en node u. • Besøk noden, og gå deretter til hver ubesøkt nabo rekursivt. • DFS rekursiv går automatisk tilbake når alle naboene til en node er besøkt. Algoritme: Procedure DFSVisit(G, u, visited) add u to visited for (u, v) ∈ E do if v not in visited then DFSVisit(G, v, visited) Fullt DFS (håndtering av flere komponenter): Procedure DFSFull(G) visited ← empty set for v ∈ V do if v not in visited then DFSVisit(G, v, visited) Forklaring: • DFSVisit besøker alle noder som er i samme komponent som startnoden u. • DFSFull sørger for at alle komponenter i grafen blir besøkt. • Når grafen representeres med nabolister, tar det O(deg(u)) tid å gå gjennom naboene til en node. • Siden hver kant blir sjekket én gang totalt, blir den totale kjøretiden: O(|V|+|E|)

2. Iterativt DFS (med eksplisitt stack) Idé: • DFS kan også implementeres uten rekursjon. • Bruk en stack for å simulere den rekursive prosessen. • Prosessen "popper" noden fra stacken, besøker den hvis den ikke er besøkt, og legger alle naboene på stacken. Algoritme: Procedure DFSIterative(G, s, visited) stack ← [s] while stack is not empty do u ← stack.pop() if u not in visited then add u to visited for (u, v) ∈ E do stack.push(v) Forklaring: • Stacken holder nodene som fortsatt har ubesøkte naboer. • Funksjonaliteten er identisk med rekursiv DFS, men her er det eksplisitt lagring i stacken. • Denne metoden unngår begrensninger i språk som har liten rekursjonsdybde. Kjøretidsanalyse: • Som rekursiv DFS: O(|V| + |E|) • Hver node og hver kant behandles én gang.

Fordeler med DFS: • Enkel å implementere og rask for grafer med moderate størrelser. • Lav minnebruk ved rekursiv implementasjon (kun én sti lagres). • Brukes til mange grafproblemer: ○ Oppdaging av komponenter ○ Sykluser og DAG-analyse ○ Topologisk sortering

Bredde-først søk (BFS) BFS er en metode for å traversere en graf ved å besøke noder lagvis, altså fra nærmeste til fjerneste fra en startnode.

1. Intuisjon • Start fra en node s. • Besøk alle direkte naboer først. • Deretter besøkes naboene til naboene, og så videre. • BFS går altså gjennom grafen nivå for nivå, i motsetning til DFS som går dypt først. Bruksområder: • Finne korteste sti fra en startnode til alle andre noder i urettet graf (og vekter like 1). • Oppdage komponenter i en graf. • Finne sykluser eller nivåer i tre- eller grafstrukturer.

2. Implementasjon BFS fra én node: Procedure BFSVisit(G, s, visited) add s to visited queue ← [s] # Start med køen som inneholder s while queue is not empty do u ← queue.dequeue() for (u, v) ∈ E do if v not in visited then add v to visited queue.enqueue(v) Full BFS (håndtering av flere komponenter): Procedure BFSFull(G) visited ← empty set for v ∈ V do if v not in visited then BFSVisit(G, v, visited)

3. Forklaring • BFSVisit besøker alle noder i komponenten til startnoden s. • BFSFull sørger for at alle komponenter i grafen blir besøkt. • Hver node legges én gang på køen, og alle naboer blir sjekket. • Når grafen representeres med nabolister, tar det O(deg(u)) tid å gå gjennom naboene til en node. • Siden hver kant sjekkes én gang totalt, blir den totale kjøretiden: O(|V|+|E|)

4. Sammenligning med DFS Egenskap DFS BFS Strategi Dypt først Lagvis/nærmeste først Datastruktur Rekursiv stakk / stack Kø Minnebruk Lav (rekursiv) Høyere, må lagre ett nivå Korteste sti Nei Ja (i uvektet graf) Bruk Komponenter, sykluser, topologisk sort Nivåer, korteste sti, komponenter

Oppsummering • BFS er iterativ og bruker en kø for å besøke noder i riktig rekkefølge. • Traverserer alle noder og kanter i grafen én gang. • Total kjøretid: O(|V| + |E|), som er optimal for urettede og rettede grafer uten vekter. • BFS er spesielt nyttig når man ønsker korteste avstand i uvektede grafer.

Topologisk sortering Topologisk sortering gjelder for rettede asykliske grafer (DAGs), og innebærer å ordne nodene i en lineær rekkefølge slik at for hver kant (u,v) kommer u før v i sorteringen. Dette brukes typisk når man har avhengigheter, for eksempel: • Oppgaver som må gjøres i en bestemt rekkefølge • Prosjekter med deloppgaver og avhengigheter • Kurs med forkunnskaper

1. Intuisjon og eksempel • Tenk på en DAG der nodene representerer hendelser og kantene representerer «må skje før». • En topologisk sortering gir et mulig gjennomføringsforløp som oppfyller alle avhengigheter. Eksempel fra kurs: Noder: emner ved IFI Kanter: forkunnskaper • Topologisk ordning 1: IN1150 → IN2080 → IN1000 → IN2040 → IN1010 → IN2010 → IN3130 → IN3040 • Topologisk ordning 2: IN1000 → IN1150 → IN1010 → IN2040 → IN2010 → IN2080 → IN3130 → IN3040 Merk: • Flere topologiske ordninger kan eksistere for samme graf. • Hver rekkefølge tilfredsstiller at alle forkunnskaper er møtt før et emne tas.

2. Algoritme (Kahn’s algoritme) Konseptuelt enkel metode: 1. Finn en node med inngrad = 0 (ingen innkommende kanter). 2. Legg noden til i den topologiske sorteringen og fjern den og alle utgående kanter fra grafen. 3. Oppdater inngradene til nodene som var naboer. 4. Gjenta prosessen til alle noder er sortert. 5. Hvis ingen node med inngrad 0 finnes, inneholder grafen en sykel, og topologisk sortering er umulig. Pseudokode: Procedure TopologicalSort(G) L ← empty list # Sortert liste S ← set of nodes with indegree 0 while S is not empty do n ← remove a node from S add n to L for each node m with an edge n → m do remove edge n → m if m has no other incoming edges then

add m to S if edges remain in G then error "Graph has at least one cycle" else return L Kjøretid: • Hver node og kant sjekkes én gang → O(|V| + |E|)

3. Bruksområder • Planlegging av prosjekter med avhengigheter • Kompilering av kode med avhengige moduler • Oppgavesett med forkunnskapskrav • Alle tilfeller der man må oppfylle "før"-betingelser

Topologisk sortering – Implementasjon Topologisk sortering kan implementeres på flere måter, men de to mest brukte metodene er: 1. Kahn's algoritme – basert på inngrad (iterativ). 2. DFS-basert topologisk sortering – basert på dybde-først søk (rekursiv).

1. Kahn's algoritme (Iterativ med stack) Ide: • Start med alle noder som har inngrad 0 (ingen innkommende kanter). • Legg disse i en stack. • Gjenta prosessen: ta ut en node, legg den i resultatlisten, fjern alle utgående kanter. • Nye noder med inngrad 0 legges til stacken. • Hvis stacken blir tom og ikke alle noder er behandlet → grafen har en sykel → topologisk sortering er ikke mulig. Pseudokode: Procedure TopSort(G) stack ← empty stack output ← empty list for v ∈ V do if indegree(v) == 0 then stack.push(v) while stack is not empty do u ← stack.pop() append u to output for each edge (u, v) ∈ E do remove edge (u, v) if indegree(v) == 0 then stack.push(v) if |output| < |V| then error "G contains a cycle" return output Kjøretid: • Hver node og kant besøkes én gang → O(|V| + |E|) • Passer godt for DAGs som representerer prosesser med avhengigheter.

2. DFS-basert topologisk sortering Idé: • Bruk et dybde-først søk på grafen. • Noden legges på en stack først etter at alle dens naboer er prosessert. • Til slutt popper vi nodene fra stacken for å få en topologisk ordning. Intuisjon: • Noder med utgrad 0 (ingen utgående kanter) blir lagt først på stacken. • Når rekursjonen backtracker, blir nodene med avhengigheter lagt senere på stacken. • Uansett hvilken node man starter DFS fra, vil alle nodene i dens komponent bli korrekt sortert. Pseudokode: Procedure DFSTopSort(G) stack ← empty stack visited ← empty set for u ∈ V do if u ∉ visited then DFSVisit(G, u, visited, stack) return stack Procedure DFSVisit(G, u, visited, stack) add u to visited for each edge (u, v) ∈ E do if v ∉ visited then DFSVisit(G, v, visited, stack) stack.push(u) Kjøretid: • Hver node og kant besøkes én gang → O(|V| + |E|) Fordeler med DFS-metoden: • Enkel å implementere rekursivt. • Naturlig for DAGs som er dypt hierarkiske.

Sammenligning Metode Bruk Fordeler Ulemper Kahn's algoritme Iterativ Enkel å forstå, lett å oppdage sykluser Krever vedlikehold av inngrad DFS Rekursiv Naturlig med dybde-først søk, kan enkelt brukes i andre grafalgoritmer Rekursjon kan bli dyp for store grafer