Vektede grafer, korteste stier og minimale spenntrær

Oversikt I denne delen av kurset ser vi på grafer der kantene har vekter, altså tall som beskriver kostnad, avstand, tid, eller andre ressurser knyttet til å gå langs kanten. Vektede grafer brukes til å modellere mange praktiske problemer: • Transportnettverk: kostnad eller tid mellom byer. • Telekommunikasjon: delay eller båndbredde mellom noder i et nettverk. • Prosjektplanlegging: kostnad for å fullføre avhengigheter mellom oppgaver. Vi fokuserer på to hovedproblemstillinger:

1. Korteste stier (Shortest Path Problem) Problemstilling: • Gitt en vektet graf og en startnode s. • Finn den billigste veien fra s til alle andre noder, eller til en bestemt node t. Typiske algoritmer: • Dijkstra – fungerer for grafer med ikke-negative vekter. • Bellman-Ford – fungerer også for grafer med negative vekter, men ingen negative sykluser. • Floyd-Warshall – finner korteste stier mellom alle par noder. Bruk: • GPS-ruteplanlegging. • Nettverksrutinger. • Prosjektplanlegging der man vil minimere tidsbruk eller kostnader.

2. Minste spenntrær (Minimum Spanning Tree, MST) Problemstilling: • Gitt en sammenhengende, urettet, vektet graf. • Finn et tre som kobler alle nodene sammen med lavest mulig total vekt. Typiske algoritmer: • Kruskal – bygger treet ved å legge til kanter i stigende vektrekkefølge, unngår sykler. • Prim – starter fra én node og utvider treet ved å alltid legge til kanten med lavest vekt som forbinder nye noder. Bruk: • Nettverksdesign (elektrisitet, vann, telekom). • Optimering av kabling og infrastruktur. • Klynging og datavisualisering.

Kort oppsummert Tema Problemstilling Vanlige algoritmer Anvendelse Korteste stier Finn billigste vei fra startnode Dijkstra, Bellman-Ford, Floyd-Warshall GPS, nettverk, prosjektplanlegging Minste spenntrær Koble alle noder med minst mulig total vekt Kruskal, Prim Infrastruktur, kabling, nettverk Merk: • Vektede grafer introduserer flere utfordringer enn uvektede grafer, fordi vi må ta hensyn til kostnader/avstander. • Algoritmene for korteste stier og MST er fundamentale i grafteori og brukes ofte i kombinasjon med andre datastrukturteknikker som prioritetskøer og union-find.

Korteste stier Målet er å finne den korteste stien i en graf fra én gitt startnode s til alle andre noder $t \in V$. • Uvektet graf: ○ Her er alle kanter like “dyre” eller “lange” – vi bryr oss bare om antall kanter. ○ Den korteste stien kan da finnes med bredde-først søk (BFS). ○ BFS besøker lagvis nodene, så den første gangen vi når en node er alltid via en kortest mulig sti (minimalt antall kanter). • Eksempel: Hvis vi har grafen med noder A, B, C, D og kanter {A-B, B-C, A-D}, vil BFS fra A først besøke B og D, deretter C.

Vektede grafer Når kantene har vekter (f.eks. tid, kostnad, avstand), fungerer ikke BFS lenger, fordi vi må ta hensyn til summen av vektene, ikke bare antall kanter. Definisjon: • En vektet graf $G=(V,E)$ har en vektfunksjon $w:E \to R$ som assosierer en numerisk verdi med hver kant. • En kant $(u,v) \in E$ har vekten $w(u,v)$. • En korteste sti fra s til t er en sti $v1,v2,\ldots,vn$ med $v1=s, vn=t$, slik at $\sum i=1n-1w(vi,vi+1)$ er minimal. • Altså: stien med lavest akkumulert vekt, ikke nødvendigvis færrest kanter. Eksempel: A --3--> B A --1--> C C --2--> B • Korteste sti fra A til B er A → C → B med total vekt 3 (1 + 2), ikke direkte A → B med vekt 3.

Algoritmer for korteste stier i vektede grafer

1. Dijkstra-algoritmen ○ Brukes når alle kantvekter er ikke-negative. ○ Holder styr på minste kjente avstand fra startnode til alle andre noder, og oppdaterer den gradvis. ○ Fungerer for både rettede og urettede grafer.

2. Bellman-Ford-algoritmen ○ Kan håndtere negative kantvekter, men grafen må ikke ha negative sykler. ○ Oppdaterer avstander gjentatte ganger, og kan oppdage negative sykler. Merk: • Begge algoritmene finner korteste stier fra én startnode til alle andre. • Valg av algoritme avhenger av om grafen har negative vekter eller ikke.

Dijkstras algoritme for korteste stier Dijkstras algoritme brukes for å finne korteste stier fra én startnode til alle andre noder i en vektet graf med ikke-negative kanter. Hovedidé • Algoritmen er en modifikasjon av bredde-først søk (BFS), men tar hensyn til vektene på kantene. • Vi ønsker alltid å besøke den nærmeste ubesøkte noden i forhold til akkumulert vekt fra startnoden. • Traversering skjer med en prioritetskø som holder styr på hvilke noder som er nærmest startnoden. Initialisering • Sett avstanden fra startnode s til seg selv: dist[s]=0 • Sett avstanden fra startnode s til alle andre noder: dist[v]=∞∀v∈V,v≠s • Sett alle noder i prioritetskøen med sin avstand som prioritet.

Hovedløkken

1. Ta ut noden u med minste avstand fra køen.
2. Gå gjennom alle naboene v til u.
3. Beregn alternativ avstand via u: c=dist[u]+w(u,v)
4. Hvis c<dist[v], oppdater distansen og prioritet i køen: dist[v]←c
5. Fortsett til køen er tom.

Egenskaper • Korrekthet: Algoritmen garanterer at når en node fjernes fra køen, har vi funnet korteste sti til den noden. • Kjøretid: Avhenger av implementasjon av prioritetskø: ○ Vanlig liste: O(∣V∣2) ○ Binær heap: O((∣V∣+∣E∣)log∣V∣) • Begrensning: Kan ikke håndtere negative kantvekter.

Eksempel A --1--> B A --4--> C B --2--> C C --1--> D • Startnode: A • Initielt: dist[A]=0, dist[B]=∞, dist[C]=∞, dist[D]=∞ • Første besøk: A → B (dist[B]=1) • Neste besøk: B → C (dist[C]=3) • Deretter: C → D (dist[D]=4) Resultat: Korteste avstander fra A: dist[A]=0, dist[B]=1, dist[C]=3, dist[D]=4

Dijkstra – kjøretidsanalyse Dijkstra-algoritmen er effektiv, men kjøretiden avhenger av hvordan vi implementerer prioritetskøen og håndterer oppdatering av avstander: Antagelser • Vi antar at DecreasePriority (oppdatering av prioritetskø) tar logaritmisk tid, dvs. O(log∣V∣). • Grafen har ∣V∣ noder og ∣E∣ kanter. Analyse

1. Fjerning av noder (RemoveMin): ○ Hver node fjernes én gang fra køen. ○ Tidskostnad: ∣V∣⋅O(log∣V∣)=O(∣V∣log∣V∣)
2. Besøk av kanter: ○ Hver kant besøkes én gang i løkken. ○ For hver kant kalles DecreasePriority. ○ Tidskostnad: ∣E∣⋅O(log∣V∣)=O(∣E∣log∣V∣)
3. Totalt: O((∣V∣+∣E∣)log∣V∣) • Hvis grafen er sammenhengende, har vi typisk ∣E∣≥∣V∣−1, så vi kan forenkle til: O(∣E∣log∣V∣) Intuisjon • RemoveMin tar tid fordi vi må finne den minste avstanden i køen. • DecreasePriority tar tid fordi vi må justere plasseringen til en node når dens avstand blir oppdatert.

Dijkstra uten DecreasePriority Noen datastrukturer (f.eks. enkel heap eller binær heap uten støtte for DecreasePriority) gjør det vanskelig å oppdatere en nodes prioritet direkte. En alternativ implementasjon: Hovedidé • I stedet for å oppdatere prioritet, setter vi inn noden på nytt i køen hver gang vi finner en kortere vei. • Dette gjør at en node kan ligge flere ganger i køen, men vi sjekker alltid om den nye distansen er bedre enn den vi allerede har lagret. Algoritme

1. dist[v] = ∞ for alle noder, dist[s] = 0

2. queue = prioritetskø med (s, 0)
3. while queue ikke er tom:
4. u = RemoveMin(queue)
5. for alle naboer v av u:

```
c = dist[u] + w(u,v)
```

```
if c < dist[v]:
```

```
    dist[v] = c
```

```
    Insert(queue, (v, c))
```

Egenskaper • Kjøretid: O(|E|log|V|) i gjennomsnitt hvis grafen er sammenhengende. • Enklere implementasjon når datastruktur ikke støtter effektiv DecreasePriority. • Node kan bli fjernet flere ganger, men distansen oppdateres kun hvis den er kortere. Oppsummering Versjon Fordel Ulempe Med DecreasePriority Mindre duplisering i køen, strømlinjeformet Krever datastruktur med DecreasePriority Uten DecreasePriority Enkel implementasjon, fungerer med standard heaps Flere innsettinger i køen, kan bruke mer tid/minne

Negative vekter i grafer Problem med negative kanter • Dijkstra fungerer ikke på grafer med negative kanter. • Hvorfor? Dijkstra er en grådig algoritme: ○ Den antar at den første korteste stien til en node er den endelige løsningen. ○ Hvis en kant har negativ vekt, kan det dukke opp en "billigere sti" senere, og Dijkstra vil allerede ha låst inn feil verdi. Negative sykler • En negativ sykel er en sykel der summen av kantvektene er negativ. • På en graf med en negativ sykel: ○ Det finnes ingen endelig korteste sti, fordi man kan gå rundt sykkelen flere ganger og stadig redusere totalvekt. • Eksempel: A → B → C → A med totalvekt -2 ○ Hver runde rundt sykkelen reduserer stien fra A til B til C ytterligere.

Bellman-Ford-algoritmen Formål • Finn korteste stier fra en startnode i grafer som kan ha negative kanter. • Oppdager samtidig negative sykler. Hovedidé

1. En sti mellom to noder kan maksimalt inneholde |V| - 1 kanter uten å gå i en sykel. ○ Hvor |V| er antall noder i grafen.
2. Algoritmen itererer |V| - 1 ganger over alle kanter, og oppdaterer estimert avstand til alle noder: ○ dist[v] = min(dist[v], dist[u] + w(u,v))
3. Etter |V| - 1 iterasjoner: ○ Hvis noen avstand fortsatt kan reduseres, har grafen en negativ sykel. Egenskaper • Korrekthet: Sikrer at korteste sti til alle noder er funnet dersom ingen negative sykler eksisterer. • Oppdagelse av negative sykler: Etter siste iterasjon, hvis noen distanse kan reduseres → negativ sykel finnes. • Kjøretid: O(|V| • |E|) ○ Itererer |V| – 1 ganger over alle kanter. Intuisjon • Bellman-Ford "slipper gjennom" forbedringer steg for steg. • Hver iterasjon sprer informasjon om korteste vei til naboer. • Til slutt stabiliseres alle avstander dersom det ikke finnes negative sykler.

Bellman-Ford-algoritmen (implementasjon) Hovedidé • Bellman-Ford finner korteste stier i en graf som kan ha negative kanter. • Algoritmen oppdaterer estimert avstand for alle noder |V| – 1 ganger, hvor |V| er antall noder i grafen. • Til slutt sjekkes om noen avstander fortsatt kan reduseres → negativ sykel. Pseudokode Procedure BellmanFord(G, s) dist ← map med ∞ som default dist[s] ← 0 repeat |V| – 1 ganger: for hver kant (u, v) i E: c ← dist[u] + w(u,v) if c < dist[v]: dist[v] ← c for hver kant (u, v) i E: if dist[u] + w(u,v) < dist[v]: error "G inneholder en negativ sykel" return dist Forklaring • dist[v] holder den nåværende beste estimert avstand fra startnode s til node v. • Hver iterasjon sprer informasjon om korteste stier gjennom grafen. • Siste sjekk for negative sykler sikrer at grafen ikke har stier som kan reduseres uendelig. • Kjøretid: O(|V|·|E|). ○ Kan også tenkes som O(|V|³) hvis man representerer grafen med nabomatrise.

Korteste stier i DAGs Spesialtilfelle: rettet asyklisk graf (DAG) • I en DAG finnes ingen sykler, så vi kan beregne korteste stier mer effektivt enn med Bellman-Ford. • Hovedtrick: besøk nodene i topologisk rekkefølge. ○ Når vi når en node u, er alle noder som kan påvirke u allerede prosessert. • Dette gjør algoritmen lineær i antall noder og kanter: O(|V| + |E|). Algoritme (DAGShortestPaths) Procedure DAGShortestPaths(G, s) dist ← map med ∞ som default dist[s] ← 0 for u i TopSort(G): for hver kant (u, v) i E: c ← dist[u] + w(u, v) if c < dist[v]: dist[v] ← c return dist Forklaring • TopSort(G) gir nodene i en rekkefølge som sikrer at vi alltid prosesserer "forløpere" før deres etterfølgere. • For hver node u oppdateres alle naboer v med mulig kortere vei. • Enkel, rask og effektiv for DAGs fordi vi ikke trenger å iterere flere ganger.

Trær Definisjon • Alle trær er grafer, men ikke alle grafer er trær. • Et tre er en graf som oppfyller følgende egenskaper: 1. Sammenhengende: det finnes en sti mellom alle par av noder. 2. Urettet: kantene har ingen retning. 3. Asyklisk: ingen sykler finnes. • Et tre med |V| noder har alltid |V| – 1 kanter. • Legger man til en ekstra kant i et tre, vil det alltid skape en sykel. Skog • En graf der hver komponent er et tre kalles en skog. • En skog kan dermed bestå av flere separate trær.

Spenntrær (Spanning Trees) Definisjon • Et spenntre av en sammenhengende urettet graf G=(VG,EG) er et tre T=(VT,ET) slik at: ○ VT=VG (alle nodene fra G er med) ○ ET⊆EG (et utvalg av kantene i G brukes) • Med andre ord: et spenntre kobler alle noder med minst mulig antall kanter, uten å lage sykler. Egenskaper • Spenntrær kan brukes til å lage korteste veier mellom noder hvis kantene har vekt (se Dijkstra). • Hver graf kan ha flere forskjellige spenntrær, avhengig av hvilke kanter man velger. • Eksempler på algoritmer som finner spenntrær: ○ Prim’s algoritme ○ Kruskal’s algoritme Visuell Intuisjon • Tenk deg en graf med noder og mange kanter. • Et spenntre velger et sett med kanter som: 1. Dekker alle nodene 2. Ikke danner sykler 3. Kan ha lavest total vekt hvis grafen er vektet • Spenntrær brukes i nettverksdesign, kabling, veiutbygging, etc.

Minimale Spenntrær Definisjon • For en urettet og vektet graf G=(V,E) er et minimalt spenntre (MST) et tre T⊆G som: 1. Dekker alle noder V 2. Ikke danner sykler 3. Har lavest mulig total kantvekt • Den totale vekten til MST er summen av vektene på kantene i treet. • Eksempel på praktisk bruk: koble sammen hus med nettverkskabler så billig som mulig. Viktige punkter • Flere MSTs kan eksistere i samme graf. ○ Dette skjer f.eks. hvis flere kanter har samme vekt. • MST er alltid et spenntre, altså et tre som dekker alle noder.

Prims algoritme for minimale spenntrær Idé • Grådig algoritme: bygger MST steg for steg. • Starter med en vilkårlig node som det foreløpige treet T. • Gjentatte ganger: 1. Finn den kanten med lavest vekt som kobler en node i T med en node utenfor T 2. Legg til denne kanten og noden i T • Prosessen fortsetter til alle noder er inkludert i T. Implementasjon • Bruk en prioritetskø til å holde oversikt over hvilke noder som

kan legges til MST. • I køen prioriteres nodene etter minste kantvekt som forbinder dem med treet, ikke akkumulert vekt som i Dijkstra. • Dette sikrer at vi alltid velger den "billigste" utvidelsen til MST. Sammenligning med Dijkstra Punkt Dijkstra Prim Formål Korteste stier fra startnode Minimalt spenntre Prioritet i kø Akkumulert avstand fra start Kantvekt til treet Resultat Korteste vei til alle noder Laveste totalvekt for treet

Prims algoritme – implementasjon Formål • Finne et minimalt spenntre (MST) for en sammenhengende, vektet og urettet graf. Algoritme (høydepunkter)

1. Start med en vilkårlig node s og sett opp en prioritetskø.
2. Oppretthold et map parents som holder styr på hvilken node som kobler hver node til MST.
3. Sett noden s med prioritet 0 i køen.
4. Gjenta til køen er tom: ○ Ta ut noden med lavest prioritet (p,u). ○ Hvis u ikke allerede er i MST (parents), legg den til og koble den via p. ○ Legg alle nabokanter til u inn i køen med prioritet = kantvekt. Pseudokode Procedure Prim(G) queue ← empty priority queue parents ← empty map Insert(queue, (null, s)) with priority 0 while queue is not empty do (p, u) ← RemoveMin(queue) if u ∉ parents then parents[u] ← p for (u, v) ∈ E do Insert(queue, (u, v)) with priority w(u, v) return parents Kjøretid • Tilsvarende som Dijkstra: O(|E| · log(|V|)), forutsatt at køoperasjonene er logaritmiske.

Kruskals algoritme – minimale spenntrær Formål • Bygger MST ved å bruke en grådig tilnærming, men på en annen måte enn Prim. Hovedidé • I stedet for å bygge ett tre, starter Kruskal med en spennskog (flere trær). • Gå gjennom alle kanter sortert etter vekt: 1. Velg kanten med lavest vekt. 2. Legg den til MST hvis den ikke danner en sykel, dvs. hvis det ikke allerede finnes en sti mellom nodene. • Fortsett til alle noder er inkludert. Resultat • Hvis grafen er sammenhengende → returnerer ett spenntre. • Hvis grafen har flere komponenter → returnerer ett spenntre per komponent. Implementasjon • Bruk gjerne en Union-Find / disjoint-set datastruktur for å sjekke sykler effektivt. • Kompleksitet: O(|E| log |E|), hovedsakelig på grunn av sortering av kantene.

Sammenligning Prim vs Kruskal Punkt Prim Kruskal Start En node Alle kanter sortert Data Prioritetskø Union-Find / sorterte kanter Tilnærming Utvider tre Bygger skog → MST Best for Tette grafer Tynne grafer

Borůvkas algoritme – minimale spenntrær Formål • Finne minimale spenntrær (MST) for en urettet, vektet graf. • Fungerer også for grafer med flere komponenter, og vil da returnere ett MST per komponent. Hovedidé

1. Start med at hver node er sitt eget tre, altså en skog av noder.
2. For hvert tre i skogen: ○ Finn den billigste kanten som forbinder treet med et annet tre. ○ Legg denne kanten til MST.
3. Slå sammen trær som nå er koblet gjennom valgte kanter.
4. Gjenta trinn 2–3 til det ikke finnes flere kanter som forbinder forskjellige trær. Egenskaper • Algoritmen er grådig, likt Kruskal og Prim. • Hver iterasjon halverer antall trær i beste fall, siden minst én kant fra hvert tre kobles til et annet. • Terminerer når alle noder i hver komponent er samlet i ett tre. • Effektiv når man har store, sparsomt koblede grafer, spesielt i parallelle eller distribuerte systemer. Visualisering • Start: noder A, B, C, D → hver er eget tre. • Iterasjon 1: legg til billigste kant fra hvert tre → noen trær kobles sammen. • Iterasjon 2: gjenta → alle trær smelter sammen til ett MST per komponent. Fordeler • Enkel å parallellisere: hvert tre kan prosessere sin billigste kant uavhengig. • Gir garanti for minimalt totalt vekt MST. • Grei å kombinere med andre MST-algoritmer i hybridløsninger. Sammenligning med Prim/Kruskal Punkt Borůvka Prim Kruskal

Start Hver node = eget tre En node Alle kanter sortert Iterasjon Legg billigste kant per tre Legg billigste kant fra MST Legg billigste kant globalt uten sykel Parallelisering Høy Lav Moderat Kompleksitet O(E log