

INF2220 - Algoritmer og datastrukturer

HØSTEN 2017

Institutt for informatikk, Universitetet i Oslo

Forelesning 6:
Grafer II

Dijkstra fort.

Minimale spenntreer

- Prim
- Kruskal

Dybde-først søk

- Løkkeleting
- DFS Spennetre

Dijkstra (fort.)

Dijkstras algoritme

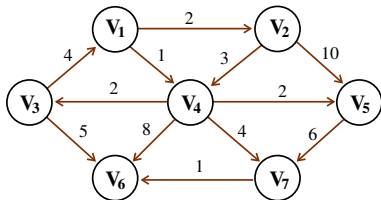
- ① For alle noder:
Sett avstanden fra startnoden s lik ∞ . Merk noden som **ukjent**
- ② Sett avstanden fra s til seg selv lik **0**
- ③ Velg en **ukjent** node v med **minimal** (aktuell) avstand fra s og marker v som **kjent**
- ④ For hver ukjent **nabonode** w til v :
Dersom avstanden vi får ved å følge veien gjennom v , er **kortere** enn den gamle avstanden til s
 - **reduserer** avstanden til s for w
 - sett **bakoverpekeren** i w til v
- ⑤ Akkurat som for uvektede grafer, ser vi bare etter **potensielle forbedringer** for naboer (w) som ennå ikke er valgt (kjent)

$$\text{uvektet: } d_w = d_v + 1 \quad \text{hvis } d_w = \infty$$

$$\text{vektet: } d_w = d_v + c_{v,w} \quad \text{hvis } d_v + c_{v,w} < d_w$$

Oppgave

Bruk Dijkstras algoritme, og fyll ut tabellen nedenfor!



Initielt:

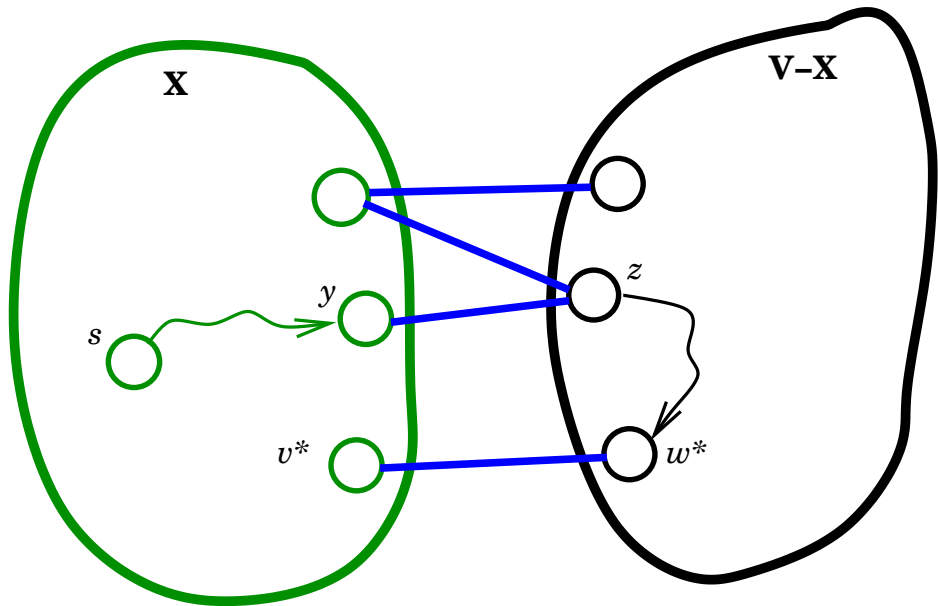
v	kjent	avstand	vei
V ₁	F	0	0
V ₂	F	∞	0
V ₃	F	∞	0
V ₄	F	∞	0
V ₅	F	∞	0
V ₆	F	∞	0
V ₇	F	∞	0

v	kjent	avstand	vei
V ₁			
V ₂			
V ₃			
V ₄			
V ₅			
V ₆			
V ₇			

Hvorfor virker algoritmen?

Invariant

- Ingen kjent node har større avstand til s enn en ukjent node
 - Alle kjente noder har riktig korteste vei satt
-
- Vi plukker ut en ukjent node v med minst avstand (d_v), markerer den som kjent og påstår at avstanden til v er riktig
 - Denne påstanden holder fordi:
 - d_v er den korteste veien ved å bruke bare kjente noder
 - de kjente nodene har riktig korteste vei satt
 - en vei til v som er kortere enn d_v , må nødvendigvis forlate mengden av kjente noder et sted,
men d_v er allerede den korteste veien fra kjente noder til v
 - Dette argumentet holder fordi vi ikke har **negative** kanter!



Tidsforbruk

- Hvis vi **leter** sekvensielt etter den ukjente noden med minst avstand tar dette $\mathcal{O}(|\mathbf{V}|)$ tid, noe som gjøres $|\mathbf{V}|$ ganger, så total tid for å finne minste avstand blir $\mathcal{O}(|\mathbf{V}|^2)$
- I tillegg oppdateres **avstandene**, maksimalt en oppdatering per kant, dvs. til sammen $\mathcal{O}(|\mathbf{E}|)$

Totalt tidsforbruk

$$\mathcal{O}(|\mathbf{E}| + |\mathbf{V}|^2) = \mathcal{O}(|\mathbf{V}|^2)$$

Raskere implementasjon (for tynne grafer):

- Bruker en **prioritetskø** til å ta vare på ukjente noder med avstand mindre enn ∞
- Prioriteten til ukjent node forandres hvis vi finner kortere vei til noden
- **deleteMin** og **decreaseKey** bruker $\mathcal{O}(\log |\mathbf{V}|)$ tid (kap. 6)

Totalt tidsforbruk

Hva med negative kanter?

En mulig løsning:

- Nodene er ikke lenger **kjente** eller **ukjente**
- Vi har i stedet en **kø** som inneholder noder som har fått forbedret avstandsverdien sin
- Løkken i algoritmen gjør følgende:
 - 1 Ta ut en node **v** fra køen
 - 2 For hver etterfølger **w**, sjekk om vi får en forbedring ($d_w > d_v + c_{v,w}$)
 - 3 Oppdater i så fall avstanden, og plasser **w** (**tilbake**)! i køen (hvis den ikke er der allerede)
- Tidsforbruket blir $\mathcal{O}(|E| \cdot |V|)$
- Det finnes ingen korteste vei med **negative løkker** i G . Det er det hvis og bare hvis samme node blir tatt ut av køen mer enn $|V|$ ganger. Da må vi **terminere** algoritmen!

Hva med asykliske grafer?

- Lineær tid ved å behandle nodene i en topologisk rekkefølge
 $\mathcal{O}(|\mathbf{E}| + |\mathbf{V}|)$
- når en node \mathbf{v} er valgt, kan $\mathbf{d}_{\mathbf{v}}$ ikke lenger senkes siden det er ingen innkommende kanter som kommer fra ukjente noder

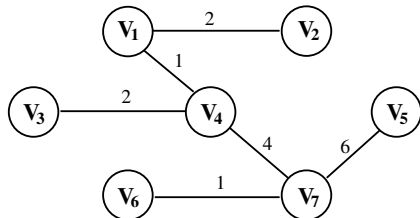
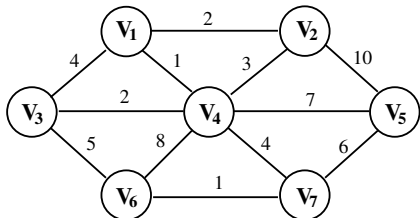
Minimale spenntær

Minimale spenntre

Definition

Et minimalt spenntre for en **urettet** graf G er et tre med kanter fra grafen, slik at alle nodene i G er forbundet til lavest mulig kostnad

- eksisterer bare for sammenhengende grafer
- Generelt: flere minimale spenntreer for samme graf

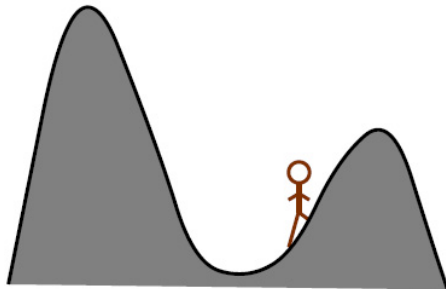


Hvor mange kanter får spenntreet?

Grådige algoritmer

Grådige algoritmer

- Prøver i hvert trinn å gjøre det som ser best ut der og da
- Typisk eksempel: Gi vekslepenger
- Raske algoritmer, men kan ikke løse alle problemer:

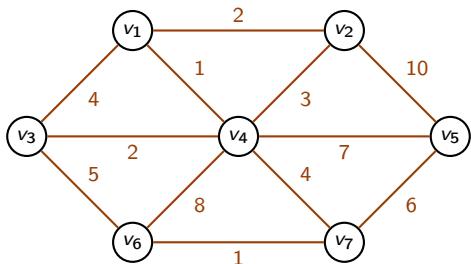


Finn det høyeste punktet!

Vi skal se på to ulike grådige algoritmer for å finne minimale spenntreer

Prims algoritme

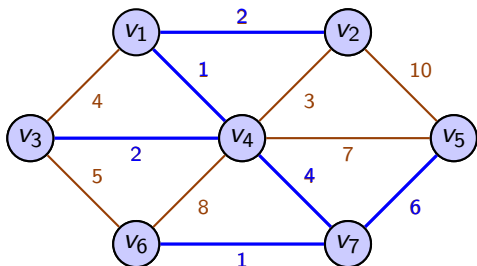
- Treet bygges opp **trinnvis**
I hvert trinn: pluss en kant (og dermed en tilhørende node) til treet
- **2 typer** noder:
 - Noder som er med i treet
 - Noder som ikke er med i treet
- Nye noder: velge en kant (u, v) med **minst** vekt slik at u er med i treet, og v ikke er det.



- Algoritmen begynner med å velge en vilkårlig node

Eksempel: La oss velge v_1

Prim: growing a tree (1)



minste kant ut fra v_1 går til v_4 , så vi legger den inn i spenntreet

- Vi har nå to mulige fortsettelser:

- Kanten fra v_1 til v_2
- Kanten fra v_4 til v_3

- Samme hvilken vi velger, vil den andre av dem bli neste kant, så vi legger dem begge inn i spenntreet

- Minste kant ut fra spenntreet går nå fra v_4 til v_7
- Så får vi kanten fra v_7 til v_6
- Til slutt får vi kanten fra v_7 til v_5

- Prims algoritme: essensielt lik Dijkstras algo for korteste vei!
 - I Prims algoritme er avstanden til en ukjent node v den minste vekten til en kant som forbinder v med en **kjent** node
- Husk: vi har urettede grafer, så hver kant befinner seg i 2 nabolister
- Kjøretidsanalysen er den samme som for Dijkstras algoritme

Hvorfor virker Prim?

Lemma (Løkke-lemma for spenntre)

Anta at \mathbf{U} er et spenntre for en graf, og at kanten e ikke er med i treet \mathbf{U} . Hvis vi legger kanten e til treet \mathbf{U} , dannes en entydig bestemt enkel løkke. Hvis vi fjerner en vilkårlig kant i denne løkken, vil vi igjen ha et spenntre for grafen.

Invariant

Det treet \mathbf{T} som dannes av de kantene (og deres endenoder) vi til nå har plukket ut, er slik at det finnes et minimalt spenntre \mathbf{U} for grafen som inneholder (alle kantene i) \mathbf{T} .

Kruskals algoritme:

Se på **kantene** en etter en, **sortert** etter minst vekt

Kanten aksepteres hvis, og bare hvis, den ikke fører til noen **løkke**

Algoritmen implementeres vha. en prioritetskø og disjunkte mengder:

- Initielt plasseres kantene i en prioritetskø og nodene i hver sin disjunkte mengde (slik at **find(v)** gir mengden til (**v**)).

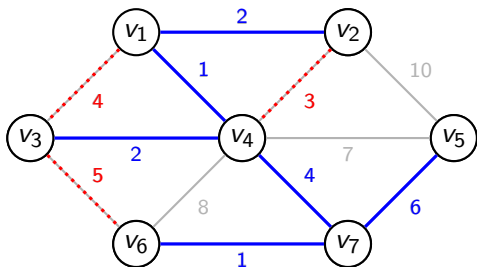
Invariant:

De disjunkte mengdene er subtrær av det endelige spennetreet

- **deleteMin** gir neste kant (**u, v**) som skal testes
 - Hvis **find(u) != find(v)**, har vi en ny kant i treet og gjør **union(u, v)**
 - Hvis ikke, ville (**u, v**) ha dannet en løkke, så kanten forkastes
- Algoritmen terminerer når prioritetskøen er tom, eventuelt når vi har lagt inn $|V| - 1$ kanter

Growing a forest (1)

Utgangspunkt for Kruskals algoritme:



Vi har to kanter med **vekt 1** — De blir til to subtrær i spenn-treet
 Vi har to kanter med **vekt 2** — De blir del av det øverste subtreet
 kant med **vekt 3**: — Den vil lage en løkke og må forkastes
 2 kanter med **vekt 4** hvor den mellom v_1 og v_3 danner en løkke, mens den andre binder de to subtrærne sammen
 kant med **vekt 5**: løkke kant med **vekt 6**: knytter den siste noden til treet

- Nå har vi lagt inn $|V| - 1$ kanter i spenn-treet og

Tidsanalyse:

- Hovedløkken går $|\mathbf{E}|$ ganger
- I hver iterasjon gjøres en **deleteMin**, to **find** og en **union**, med samlet tidsforbruk

$$\mathcal{O}(\log |\mathbf{E}|) + 2 \cdot \mathcal{O}(\log |\mathbf{V}|) + \mathcal{O}(1) = \mathcal{O}(\log |\mathbf{V}|)$$

(fordi $\log |\mathbf{E}| < 2 \cdot \log |\mathbf{V}|$)

- Totalt tidsforbruk er $\mathcal{O}(|\mathbf{E}| \cdot \log |\mathbf{V}|)$

Prim vs. Kruskal

- Prims algoritme er noe mer effektiv enn Kruskals, spesielt for tette grafer
- Prims algoritme virker bare i sammenhengende grafer
- Kruskals algoritme gir et minimalt spenn-tre i hver sammenhengskomponent i grafen

Oppgave

Let T be a minimum spanning tree of G . Then, for any pair of vertices s and w , the shortest path from s to w in G is the path from s to w in T .

- 1 True
- 2 False

Oppgave

Let w be the minimum weight among all edge weights in an undirected connected graph. Let e be a specific edge of weight w . Which of the following is FALSE?

- 1 Every minimum spanning tree has an edge of weight w
- 2 If e is not in a MST T , all edges have the same weight in the cycle formed by adding e to T
- 3 e is present in every minimum spanning tree
- 4 There is a minimum spanning tree containing e

Dybde-først søk

Dybde-først søk

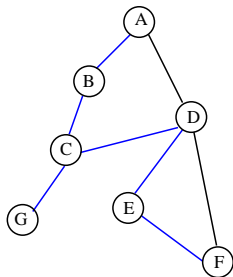
- klassisk graf traversering
 - generalisering av **prefiks** traversering for trær
 - gitt: start node v : **rekursivt** traverserer alle nabonodene
 - rekursjon \Rightarrow vi undersøker alle noder som kan nåes fra første etterfølger til v , før vi undersøker *neste* etterfølger til v
 - for vilkårlige grafer: pass på å **terminerer** rekursjon!
- \Rightarrow unvisited \leftrightarrow visited nodes.

DFS

Initialize: all nodes “unvisited”.

- Recur:**
- when visited: return immediately
 - when unvisited
 - set to “visited”
 - recur on all neighbors

```
void dybdeFørstSøk(Node v) {  
    v.merke = true;  
    for < hver nabo w til v > {  
        if (!w.merke) {  
            dybdeFørstSøk(w);  
        }  
    }  
}
```



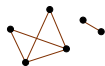
Løkkeleting

- Vi kan bruke dybde-først søk til å sjekke om en graf har løkker
- 3 verdier til tilstandsvariablen: **usett**, **igang** og **ferdig** (besøkt)

```
void løkkeLet(Node v) {
    if (v.tilstand == igang) {
        < Løkke er funnet >
    } else if (v.tilstand == usett) {
        v.tilstand = igang;
        for < hver nabo w til v > {
            løkkeLet(w);
        }
        v.tilstand = ferdig;
    }
}
```

Er grafen sammenhengende?

urettet graf & DFS



En **urettet** graf er sammenhengende hvis og bare hvis et dybde-først søk som starter i en tilfeldig node, besøker alle nodene i grafen

rettet graf & DFS



En **rettet** graf er sterkt sammenhengende hvis og bare hvis vi fra hver eneste node v klarer å besøke alle de andre nodene i grafen ved et dybde-først søk fra v

Hvis grafen **ikke** er sammenhengende, kan vi foreta nye dybde-først søk fra noder som ikke er besøkte, inntil alle nodene er behandlet

Urettet grafer

Dybde-først-spenntre

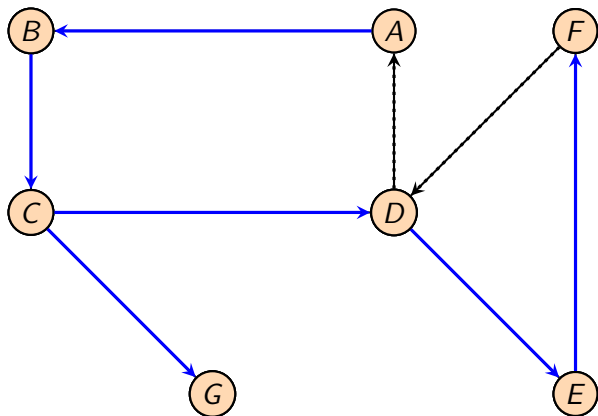
- for urettet sammenhengende grafer
- *ikke sammenhengende* grafer: dfs spanning forest
- huske “back-pointers”

Ulike type kanter

gitt en bestemt kjøring av dfs

- 1 tree edges
- 2 back edges

DFS, urettet graf (1)



Neste forelesning: 5. oktober
Grafer III:

- Biconnectivity
- Strongly connected components