Vektede grafer, korteste stier og minimale spenntrær

IN2010 – Algoritmer og Datastrukturer

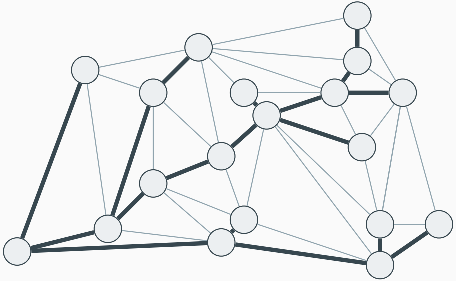
Lars Tveito

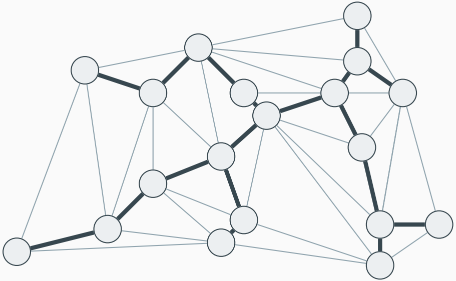
Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2024

Oversikt

Oversikt



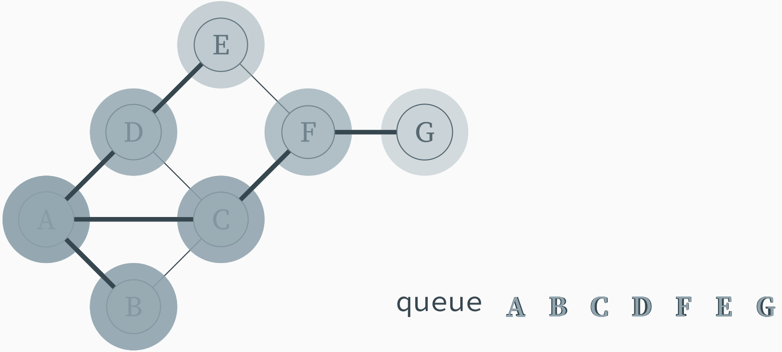


- Denne uken skal vi se på *vektede* grafer
- Hvordan finne den billigste veien fra én node til andre noder?
- Hvordan finne den billigste måten å koble alle noder i en graf?

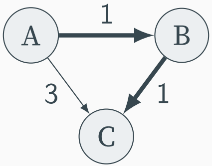
Korteste stier

Korteste stier

- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk



Vektede grafer

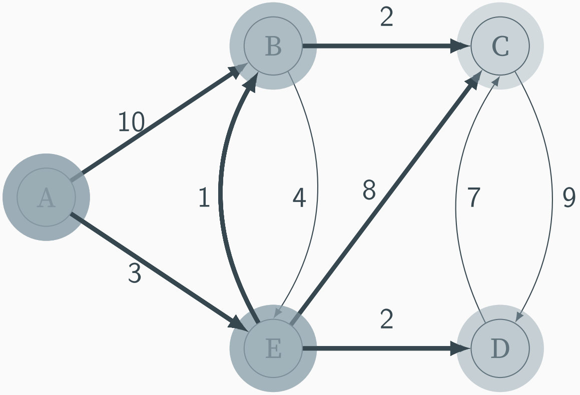


- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- En *korteste sti* mellom $s \in V$ og $t \in V$ er en sti $v_1, v_2, \ldots, v_n$ slik at
  - $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal
- Det vil si en sti som har lavest *akkumulert* vekt
- Vi skal se på to algoritmer for å finne korteste stier i vektede grafer:
  - Dijkstra: dersom vi antar at det ikke finnes kanter med negativ vekt
  - Bellman-Ford: dersom vi har kanter med negativ vekt
- Algoritmene fungerer både for rettede og urettede grafer

Dijkstras algoritme for korteste stier

- Input er en graf G og en startnode s
- Idéen er å modifisere bredde-først søk til å ta høyde for kantenes vekt
  - Vi traverserer grafen fra s med en *prioritetskø* som ordner noder etter avstand fra startnoden
  - Vi besøker alltid den «nærmeste» ubesøkte noden fra startnoden med hensyn til akkumulert vekt
- Initielt settes
  - avstanden fra startnoden s til s til 0
  - avstanden fra startnoden s til alle andre noder til ∞

Dijkstras algoritme for korteste stier (eksempel)



queue	dist
A	0
B	~~∞~~ ~~10~~ 4
C	~~∞~~ ~~11~~ 6
D	~~∞~~ 5
E	~~∞~~ 3

Dijkstra (tradisjonell implementasjon)

Algorithm: Dijkstras algoritme for korteste stier (tradisjonell)

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
Procedure Dijkstra(G, s)
    queue ← empty priority queue
    dist ← empty map
    for v ∈ V do
        dist[v] ← ∞
        Insert(queue, v) with priority ∞
    dist[s] ← 0
    DecreasePriority(queue, s, 0)

    while queue is not empty do
        u ← RemoveMin(queue)
        for (u, v) ∈ E do
            c ← dist[u] + w(u, v)
            if c < dist[v] then
                dist[v] ← c
                DecreasePriority(queue, v, c)
    return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk
- Hver node poppes av prioritetskøen én gang
- Det koster $\mathcal{O}(|V| \cdot \log(|V|))$ fordi
  - det er $|V|$ noder på prioritetskøen
  - `RemoveMin` er logaritmisk
- Hver kant besøkes nøyaktig én gang
- Det koster $\mathcal{O}(|E| \cdot \log(|V|))$ fordi
  - fra hver kant kaller vi `DecreasePriority`
  - `DecreasePriority` er logaritmisk
- Til sammen har vi $\mathcal{O}((|V| + |E|) \cdot \log(|V|))$
- Kan forenkles til $\mathcal{O}(|E| \cdot \log(|V|))$ hvis grafen er sammenhengende

```
Procedure Dijkstra(G, s)
  queue ← empty priority queue
  dist ← empty map
  for v ∈ V do
    dist[v] ← ∞
    Insert(queue, v) with priority ∞
  dist[s] ← 0
  DecreasePriority(queue, s, 0)

  while queue is not empty do
    u ← RemoveMin(queue)
    for (u, v) ∈ E do
      c ← dist[u] + w(u, v)
      if c < dist[v] then
        dist[v] ← c
        DecreasePriority(queue, v, c)
  return dist
```

Dijkstra (implementasjon uten `DecreasePriority`)

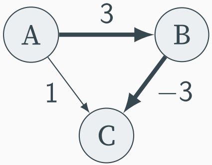
Algorithm: Dijkstras algoritme for korteste stier

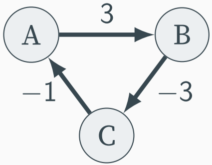
Input: En vektet og sammenhengende graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
Procedure Dijkstra(G, s)
    dist ← empty map with ∞ as default
    queue ← priority queue containing s with priority 0
    dist[s] ← 0

    while queue is not empty do
        u ← RemoveMin(queue)
        for (u, v) ∈ E do
            c ← dist[u] + w(u, v)
            if c < dist[v] then
                dist[v] ← c
                Insert(queue, v) with priority c
    return dist
```

Negative vekter





- Dijkstra kan gi feil svar for grafer med negative vekter fordi den er «grådig»
    - En grådig algoritme går ut i fra at den første løsningen er det beste
- Dersom grafen inneholder negative sykler finnes det ingen korteste sti
    - Fordi det alltid lønner seg å ta en runde til i en negativ sykel
- Bellman-Ford finner korteste sti i grafer med negative vekter
    - eller oppdager negative sykler

Bellman-Ford

- En sti kan ikke inneholde mer enn $|V| - 1$ kanter
  - En vei med $|V|$ kanter må inneholde en sykel
- Algoritmen oppdaterer estimert avstand for alle noder «mange nok» ganger
  - $|V| - 1$ er mange nok ganger!
- Hvis en node får en lavere estimert avstand etter $|V| - 1$ iterasjoner
  - så inneholder G en negativ sykel

Bellman-Ford (implementasjon)

Algorithm: Bellman-Fords algoritme for korteste stier

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnode s
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
Procedure BellmanFord(G, s)
    dist ← empty map with ∞ as default
    dist[s] = 0

    repeat |V| − 1 times
        for (u, v) ∈ E do
            c ← dist[u] + w(u, v)
            if c < dist[v] then
                dist[v] ← c

    for (u, v) ∈ E do
        c ← dist[u] + w(u, v)
        if c < dist[v] then
            error G contains a negative cycle
    return dist
```

- Denne går gjennom alle kanter $|V|$ ganger
- Det gir $\mathcal{O}(|V| \cdot |E|)$
- Dette kan forenkles til $\mathcal{O}(|V|^3)$

Korteste stier i DAGs

- Hvis G er en rettet asyklisk graf (DAG), kan vi finne korteste stier i $\mathcal{O}(|V| + |E|)$
- Korteste sti til en node $u \in V$ kan ikke påvirkes av en etterfølger $v \in V$
  - Fordi grafen ikke kan inneholde sykler
- Vi besøker nodene i *topologisk sortert rekkefølge*
  - Når $u \in V$ besøkes vet vi at alle noder som kan påvirke den er ferdig prosessert

ALGORITHM: KORTESTE STIER I EN DAG

Input: En vektet, asyklisk graf $G = (V, E)$ med vektfunksjon w og en startnode s

Output: Et map `dist` som angir korteste vei fra s til alle noder i G

```
Procedure DAGShortestPaths(G, s)
    dist ← empty map with ∞ as default
    dist[s] = 0

    for u ∈ TopSort(G) do
        for (u, v) ∈ E do
            c ← dist[u] + w(u, v)
            if c < dist[v] then
                dist[v] ← c
    return dist
```

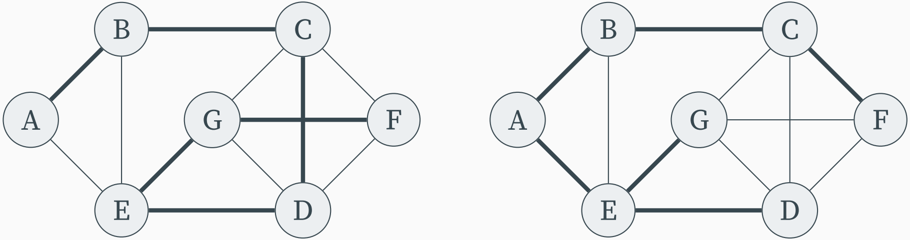
Minimale spenntrær

Trær

- Alle trær er grafer, men ikke alle grafer er trær
- En *sammenhengende*, *urettet* og *asyklisk* graf $G = (V, E)$ er et tre
  - Et slik tre har nøyaktig $|V| - 1$ kanter
  - Å legge til en kant i et tre vil føre til en sykel
- En enkel graf der hver komponent er et tre kalles en skog

Spenntrær

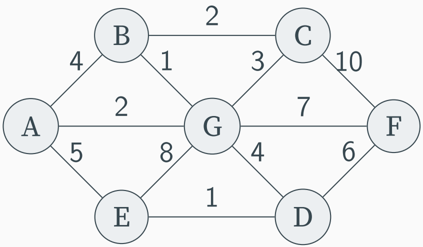
- Et *spenntre* av en sammenhengende og urettet graf $G = (V_G, E_G)$ er et tre $T = (V_T, E_T)$, der $V_T = V_G$ og $E_T \subseteq E_G$

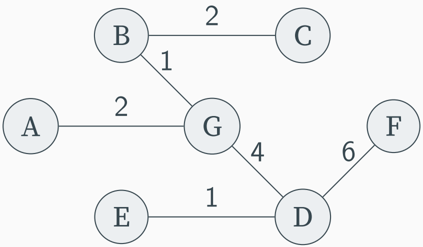


- Det vil si et tre som består av *de samme nodene* og et *utvalg* av kantene
- Vi har sett spenntrær fra bredde-først søk og Dijkstra allerede

Minimale spenntrær

- Når G er *urettet og vektet* er vi ofte interessert i å finne *minimale* spenntrær
- Et eksempel er å koble sammen hustander med nettverkskabler billigst mulig

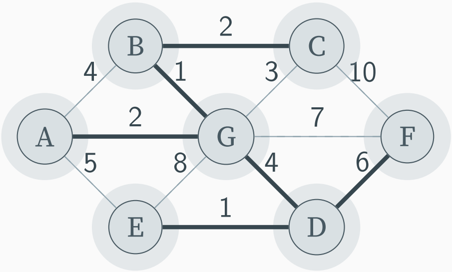


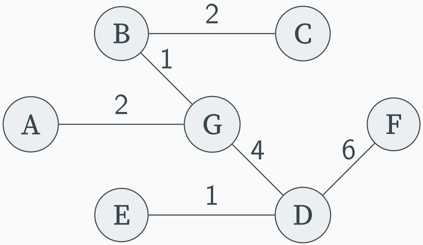


- Gitt en graf G er T et minimalt spenntre for G hvis ingen andre spenntrær for G har mindre total vekt
  - Legg merke til at det kan finnes flere minimale spenntrær for samme graf
  - (For eksempel hvis alle kantene har samme vekt)

Prims algoritme for minimale spenntrær

- Prims algoritme bygger opp et minimalt spenntre T grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet T
- Vi velger neste kant til å være den kanten med minst vekt som sammenkobler en node fra T og en node som *ikke* er i T
- I likhet med Dijkstra bruker vi en prioritetskø
- Her prioriterer vi nodene etter vekten på kanten
    - snarere enn den akkumulerte vekten av stien så langt (som vi gjorde for Dijkstra)





Prims algoritme for minimale spenntrær (implementasjon)

Algorithm: Prims algoritme for minimale spenntrær

Input: En sammenhengende, vektet, urettet graf $G = (V, E)$ med vektfunksjon w
Output: Et minimalt spenntre for G

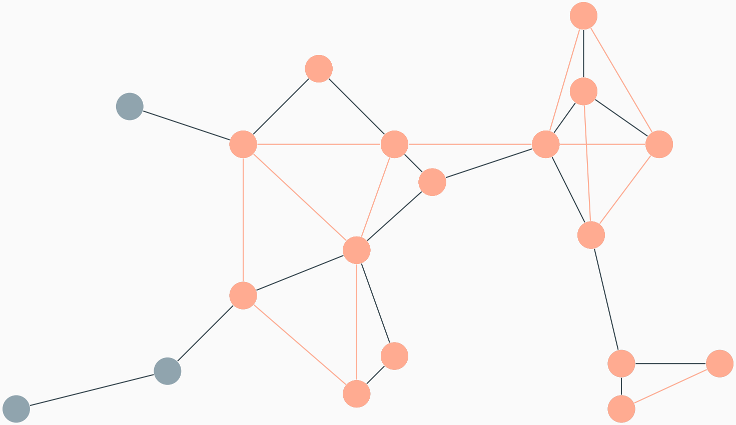
```
Procedure Prim(G)
    queue ← empty priority queue
    parents ← empty map
    Insert(queue, (null, s)) with priority 0, for some arbitrary s ∈ V
    while queue is not empty do
        (p, u) ← RemoveMin(queue)
        if u ∉ parents then
            parents[u] ← p
            for (u, v) ∈ E do
                Insert(queue, (u, v)) with priority w(u, v)
    return parents
```

- Kjøretidskompleksiteten er den samme som Dijkstra: $\mathcal{O}(|E| \cdot \log(|V|))$

Kruskals algoritme for minimale spenntrær

- Kruskals algoritme for minimale spenntrær er i grådig, i likhet med Prim
- I motsetning til Prim bygger ikke Kruskal opp ett spenntre, men en spennskog
  - En spennskog er flere trær (eller en mengde med trær)
- Hvis G er sammenhengende vil Kruskal returnere ett spenntre
- Hvis G består av flere komponenter returnerer Kruskal ett spenntre for hver komponent

Kruskals algoritme for minimale spenntrær (illustrasjon)



Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas
- I likhet med Kruskal gir den et minimalt spenntre for hver komponent i grafen
- Den går ut på å anse hver node i grafen som et eget lite tre
    - Disse trærene utgjør en skog
- For alle trærene, velg den billigste kanten som forbinder treet med et annet
- Algoritmen terminerer når det ikke lenger finnes kanter som forbinder forskjellige trær

Borůvkas (eksempel)