Oversikt—Spenntrær

IN2010 – Algoritmer og Datastrukturer

Uke 39, 2020

Institutt for Informatikk

Sammenhengende grafer

En graf $G = (V, E)$ kalles for *sammenhengende* hvis det finnes en sti mellom hvert par av noder

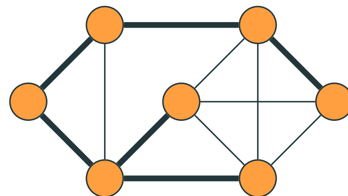
- "alle noder kan nå hverandre"
- Men hvordan forbinde nodene slik at man bruker minst mulige ressurser?
- f.eks. koble sammen hus med fibernett og bruke minst mulig kabel

Spenntrær

For en sammenhengende graf $G = (V, E)$ er et *spenntre* et tre $G_T = (V_T, E_T)$ der $V = V_T$, og $E_T \subseteq E$.

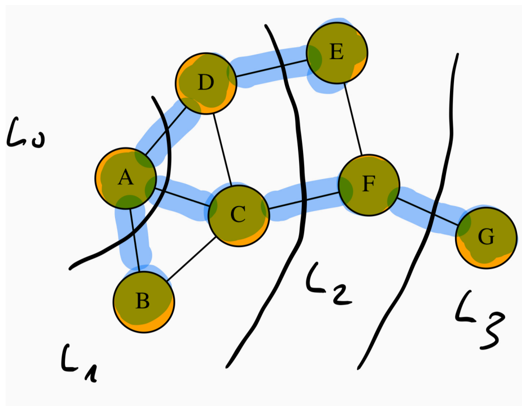
Liten påminnelse:

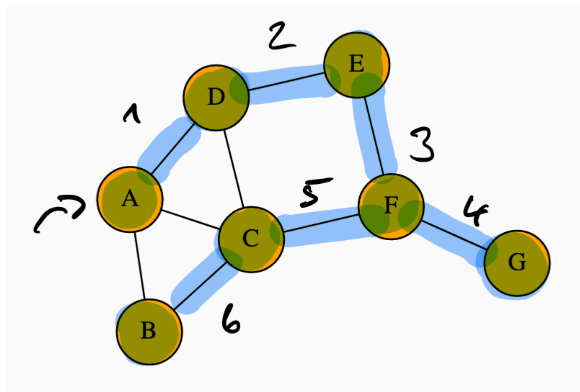
- et tre er en graf uten sykler
- trær er sammenhengende
- så et spenntre av G består av alle noder og akkurat nok kanter til å forbinde alle disse nodene
- å legge til en vilkårlig kant fra G som ikke er i E_T vil føre til en sykel



Spenntrær: Noen observasjoner

- Man kan vise: Hvis en graf har flere kanter enn noder, så inneholder grafen en sykel
- dermed har spenntrær $|V| - 1$ kanter.
- Når grafen er uvektet, så kan vi bruke DFS og BFS til å gi oss et spenntre





- men hva hvis kantene er vektet, altså hver kant e får en vekt $w(e)$? Hvordan finne *minimal spenntre*: et spenntre som har minimal vekt?

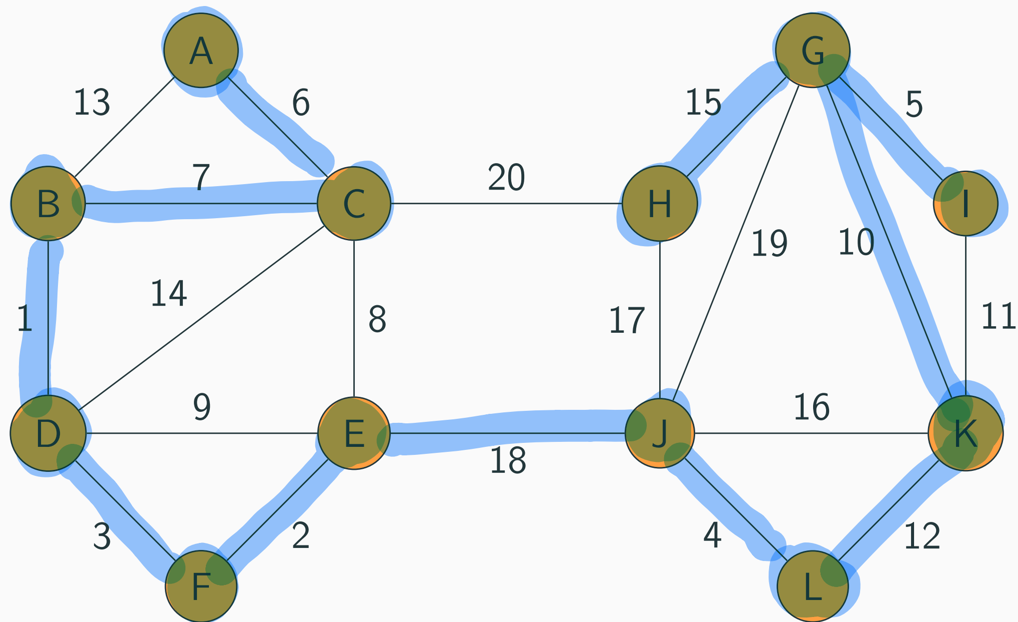
Minimale spenntrær

Vi skal se på tre såkalte "grådige" algoritmer, hvor vi :

- Prims algoritme (grådig valg av noder)
- Kruskals algoritme (grådig valg av kanter)
- Borůvkas algoritme (grådig valg av kanter, parallelliserbar)

Alle algoritmer tar en vektet, sammenhengende graf G og returnerer et minimalt spenntre T.

Prims algoritme (grådige noder)



Prims algoritme: Pseudokode

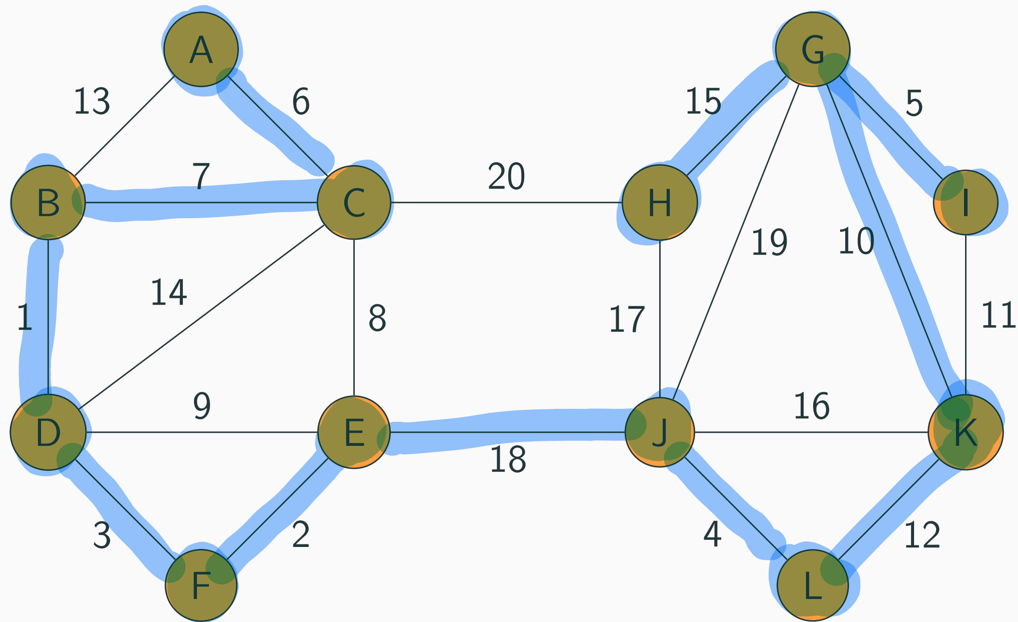
Algorithm 1: Prims algoritme

```
1  Procedure Prim(G)
2      initialize T as empty tree, Q as empty heap
3      for each vertex u in G do
4          D[u] = ∞
           // Q contains (Node, Edge), where Edge has lowest weight
              of edges from T to Node, and uses D[u] to compare
5          Q.add(((u, None), D[u]))
6      pick v ∈ V, set D[v] = 0
7      while Q not empty do
8          (s, e) = Q.removeMin()
9          add s and e to T
           // check neighbors of s in Q
10         for edge a = (s, z) with z in Q do
11             if w(a) < D[z] then
12                 D[z] = w(a)
13                 Change entry of z in Q to ((z, a), D[z])
14     return T
```

Analyse:

- 5 (for): for hver node i G, heap-insert $O(\log(|V|))$
- 8,9: while loop har $|V|$ iterasjoner, med en heap-remove $O(\log(|V|))$
- 11: gjøres totalt maksimalt $|E|$ ganger (ingen kanter gjentas). Oppdatering av verdier i Q tar $O(\log(|V|))$ tid
- **Totalt:** $O((|V| + |E|)\log(|V|))$, som er $O(|E|\log(|V|))$.

Prims algoritme (grådige noder)



Kruskals algoritme: Pseudokode

Algorithm 2: Kruskals algoritme

```
1  Procedure Kruskal(G)
2      initialize T as empty tree
3      initialize Q with all edges
4      for each vertex v in G do
5          define C(v) = {v}
6      while T has fewer than n − 1 edges do
7          (u, v) = Q.removeMin()
8          if C(u) ≠ C(v) then
9              add (u, v) to T
10             C(u), C(v) = C(u) ∪ C(v)
11     return T
```

Alt tilsammen: $O(|E|\log(|V|))$

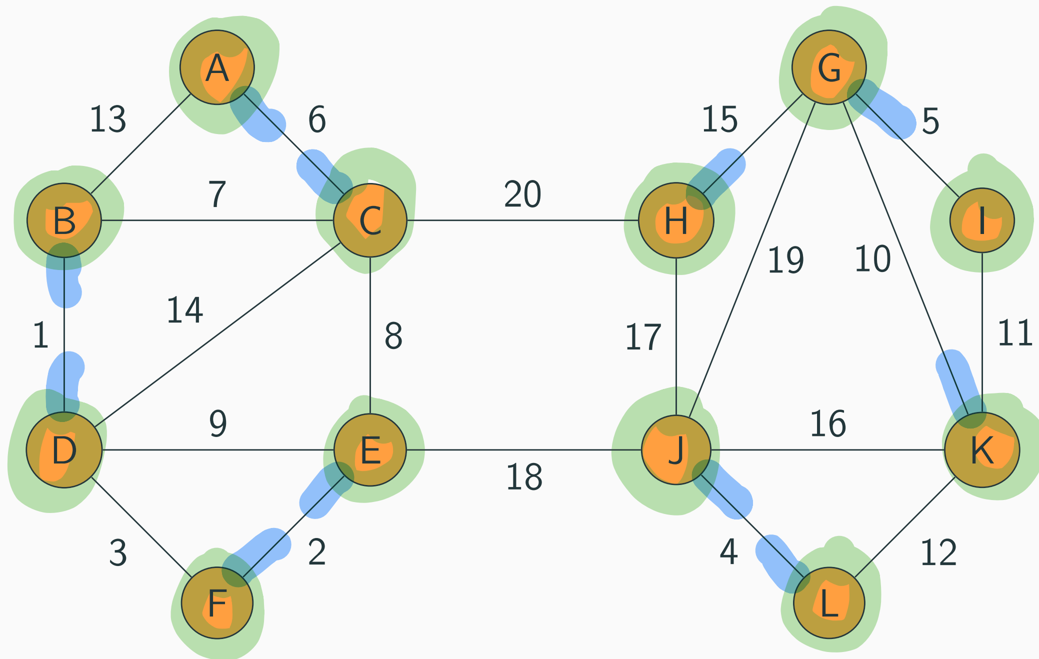
Analyse:

- 3 (initialize Q): vi kjører heap-insert ($O(log(|E|))$) for alle kanter, tilsammen $O(|E|\log(|E|))$, som er $O(|E|\log(|V|))$ (se heap-remove argumentet nederst)
- 4 (for): $|V|$ iterasjoner av konstanttid operasjoner, $O(|V|)$
- 6 (while): $|E|$ iterasjoner, med heap-remove ($O(\log(|E|))$) og cluster-merging

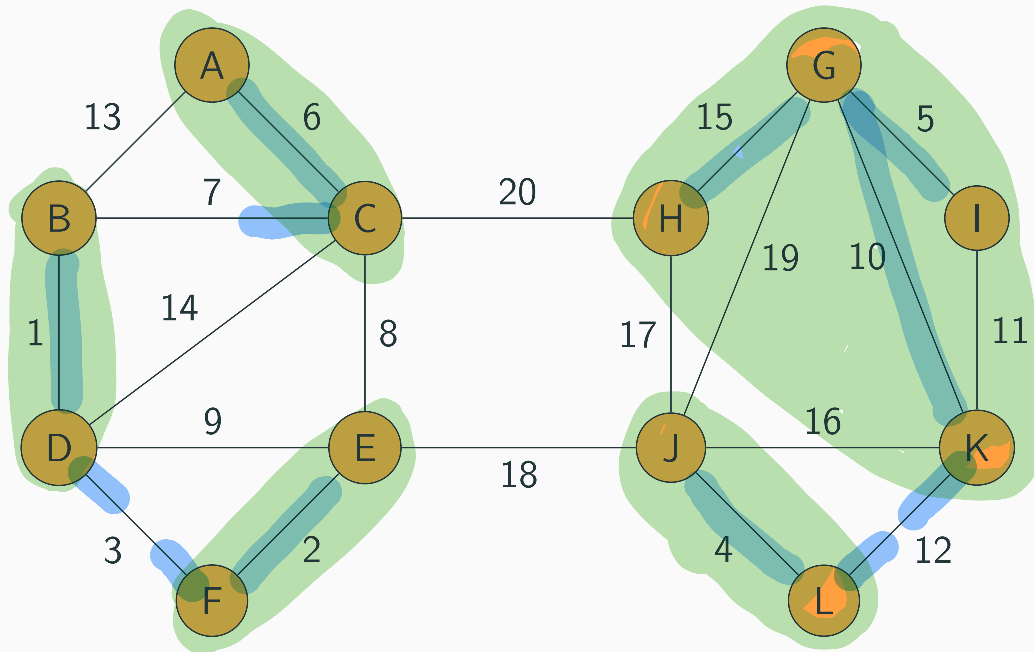
Cluster-merge: hver cluster-merge dobler størrelsen av clusteret/halverer antall clustere. Dermed: hver node merged inn i ny cluster $\log(|V|)$ ganger. Summert: $O(|V|\log(|V|))$

Heap-remove: Vi har sett at $|E|$ er $O(|V|^2)$. Så $O(\log(|E|))$ er $O(\log(|V|^2))= O(2\log(|V|))= O(log|V|)$

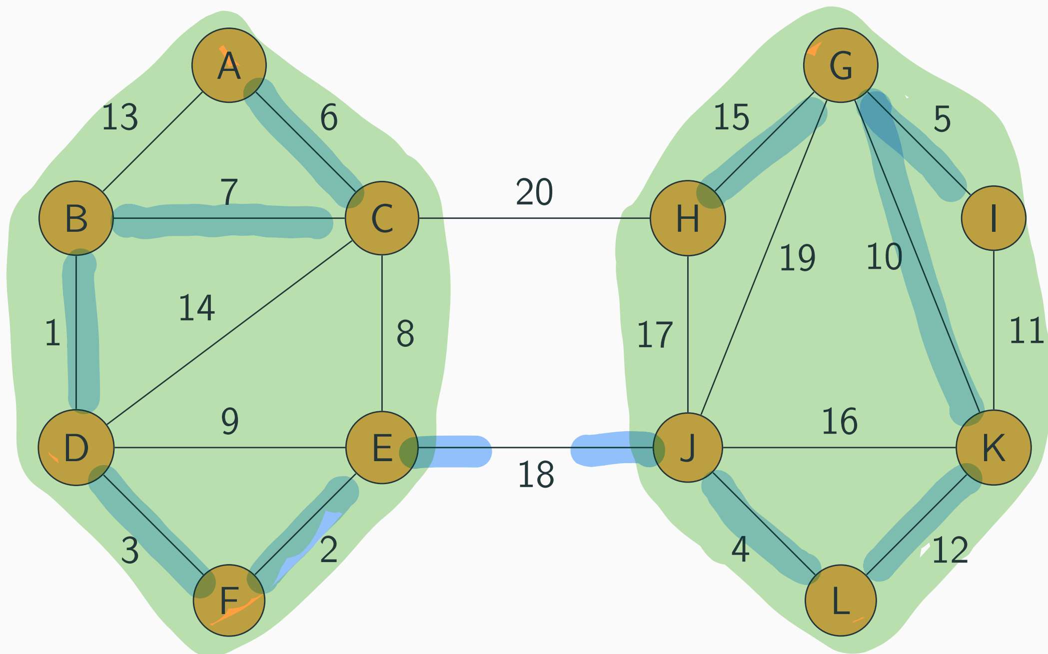
Borůvkas algoritme (grådige kanter)



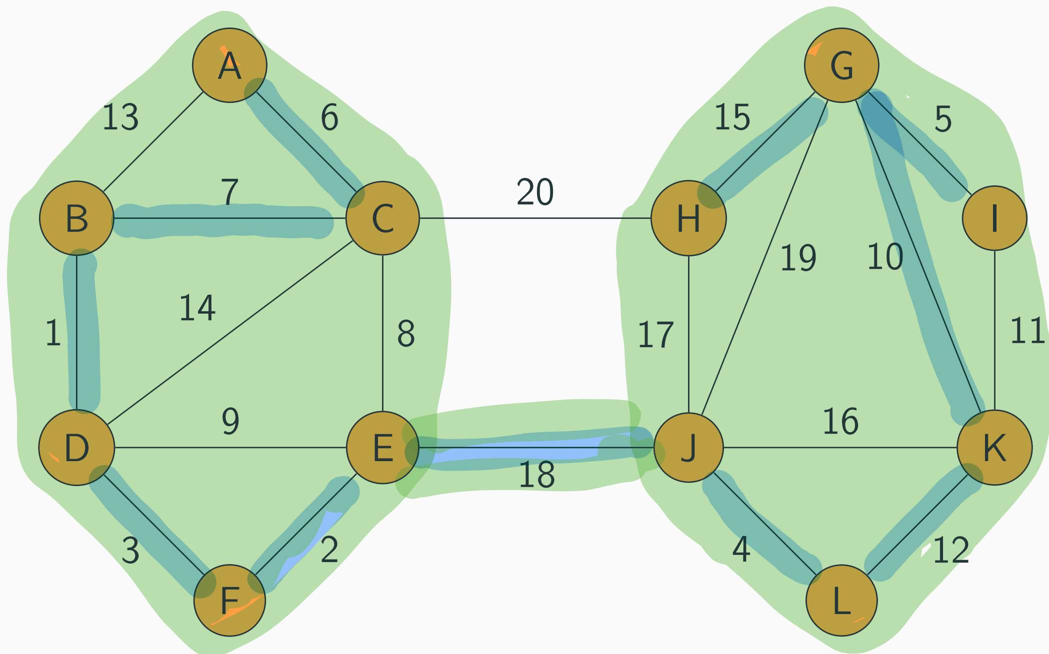
Borůvkas algoritme (grådige kanter)



Borůvkas algoritme (grådige kanter)



Borůvkas algoritme (grådige kanter)



Borůvkas algoritme: Pseudokode

Algorithm 3: Borůvkas algoritme

```
1  Procedure Boruvka(G)
2      initialize T as empty tree
3      for each vertex v in G do
4          add v to T
5      while T has more than one component do
6          for each component C in T do
7              for each vertex v in C do
8                  Comp(v) = C
9              cheapest(C) = None
10         for each edge e = (u, v) in G do
11             if Comp(u) ≠ Comp(v) then
12                 if w(e) < w(cheapest(Comp(u))) then
13                     cheapest(Comp(u)) = e
14                 if w(e) < w(cheapest(Comp(v))) then
15                     cheapest(Comp(v)) = e
16         for each cheapest(C) ≠ None do
17             add cheapest(C) to T
18     return T
```

Analyse:

- 5 (while): antall komponenter (minst) halveres hver iterasjon. Dermed er det $O(\log(|V|))$ iterasjoner.
- 6,7 (for): itererer over noder i en komponent, som er begrenset av $|V|$.
- 10 (for): $|E|$ iterasjoner, bare konstanttid kall i løkken, dermed $O(|E|)$.
- 16 (for): itererer over alle kanter, $O(|E|)$
- **Totalt:** $O((|V| + |E|)\log(|V|)) = O(|E|\log(|V|))$

Prim vs. Kruskal vs. Borůvka

Alle har samme verste tilfellet analyse...så hvilken algoritme skal man velge?

- på tynne grafer er Kruskal i praksis ofte raskere
- hvis man har tilgang til kantene sortert etter vekt: Kruskal raskere
- det er mulig å implementere Prim slik at den blir raskere (med datastruktuerer ikke dekket av pensumet), med verste tilfellet $O(|E| + |V| \log(|V|))$. Da er den mye bedre for tette grafer. **Dette er ikke pensum**
- Borůvka er lett å parallellisere