Vektede grafer, korteste stier og minimale spenntrær

IN2010 – Algoritmer og Datastrukturer

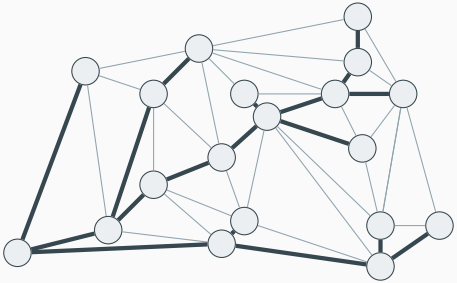
Lars Tveito

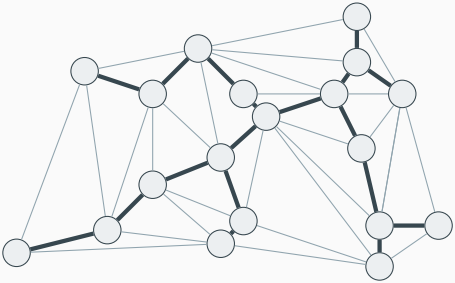
Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2022

Oversikt uke 40

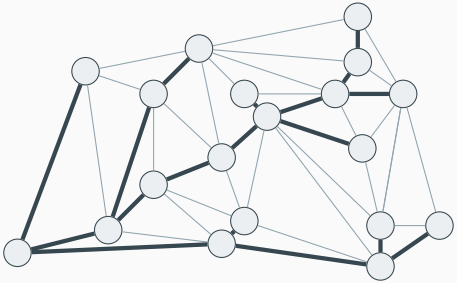
Oversikt uke 40

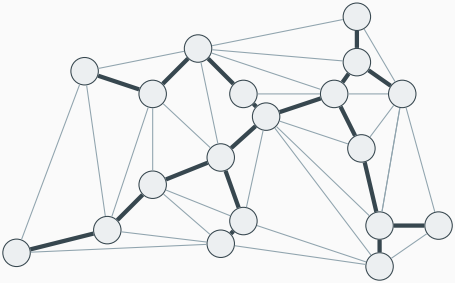




- Denne uken skal vi se på *vektede* grafer

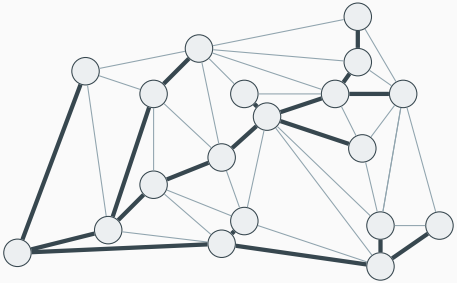
Oversikt uke 40

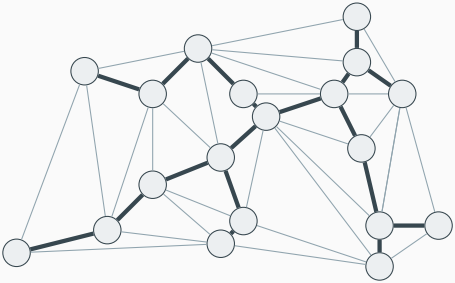




- Denne uken skal vi se på *vektede* grafer
- Hvordan finne den billigste veien fra én node til andre noder?

Oversikt uke 40





- 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

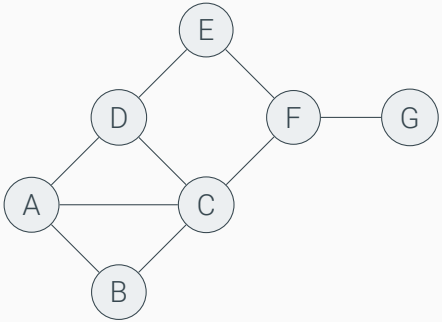
- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre

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

Korteste stier

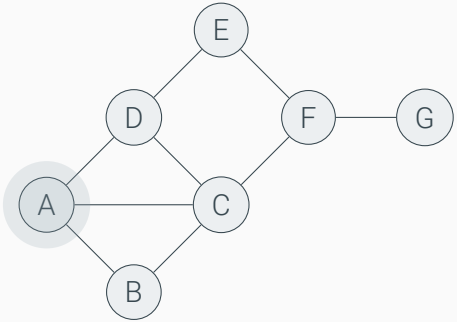
- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk



queue

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

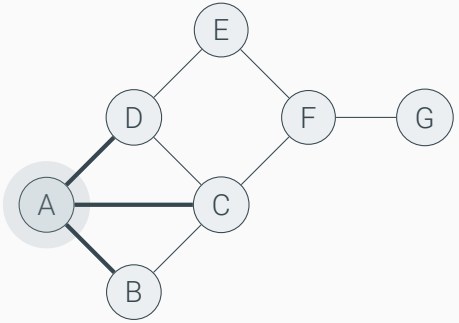


queue

A

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



queue

A

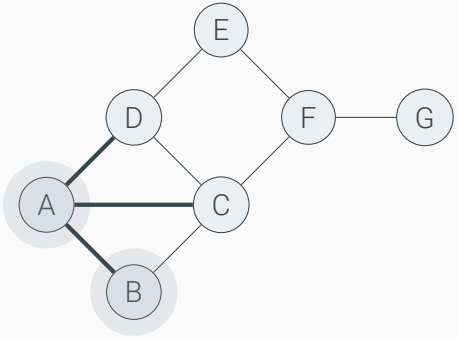
B

C

D

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



queue

A

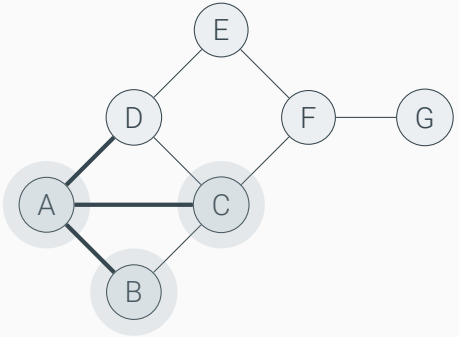
B

C

D

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



queue

A

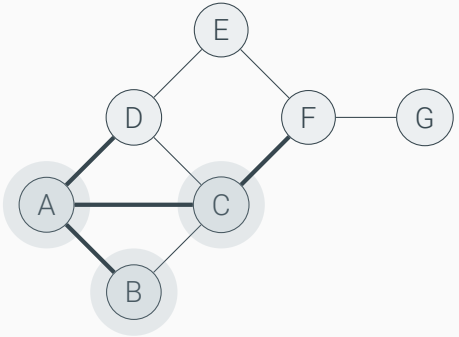
B

C

D

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



queue

A

B

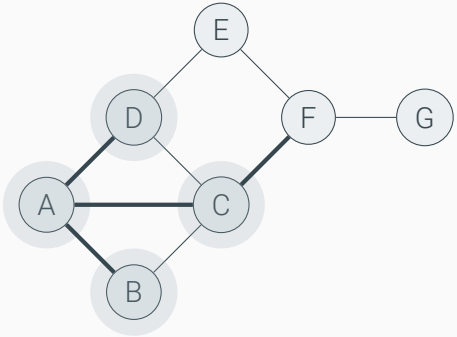
C

D

F

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

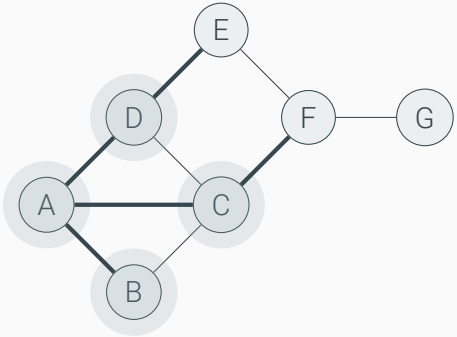


queue

A
B
C
D
F

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

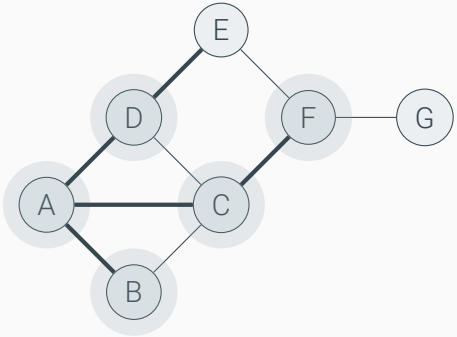


queue

A
B
C
D
F
E

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

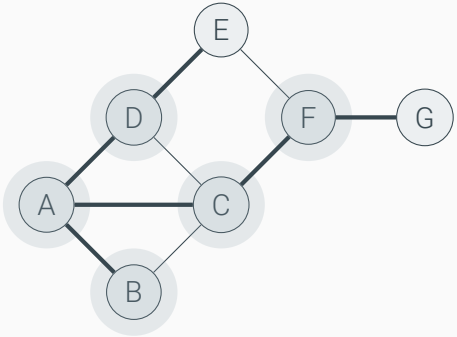


queue

A
B
C
D
F
E

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

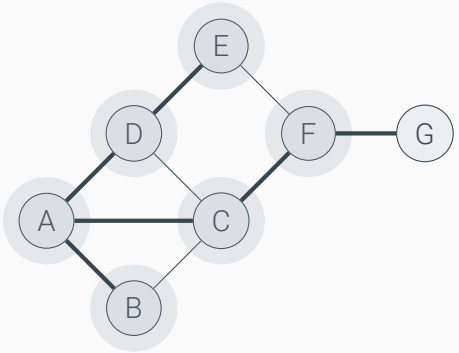


queue

A
B
C
D
F
E
G

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

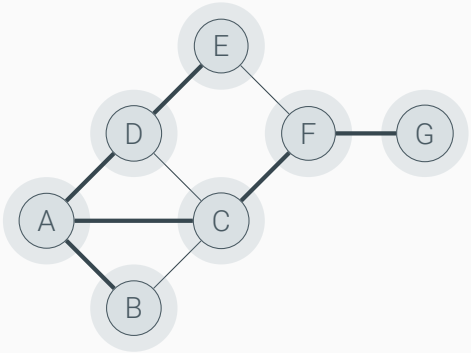


queue

A
B
C
D
F
E
G

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

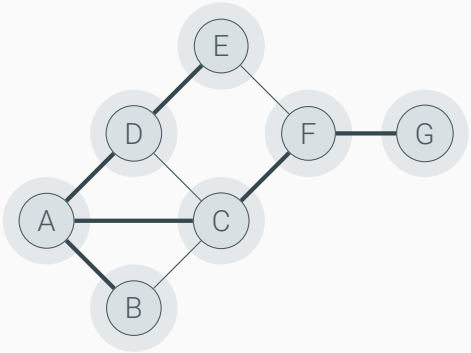


queue

A
B
C
D
F
E
G

KORTESTE STIER

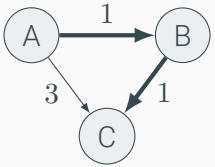
- Vi ønsker å finne den *korteste* stien i en graf fra en gitt node til alle andre
- Hvis grafen er *uvektet* kan bruke bredde-først søk



queue

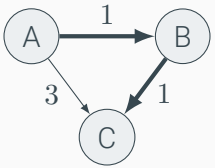
A
B
C
D
F
E
G

Vektede grafer



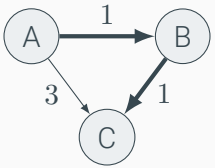
- Bredde-først søk fungerer ikke på *vektede* grafer

Vektede grafer



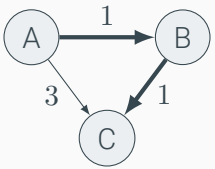
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w

Vektede grafer



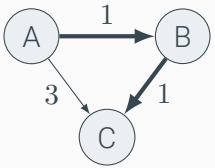
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten

Vektede grafer



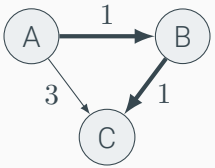
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$

Vektede grafer



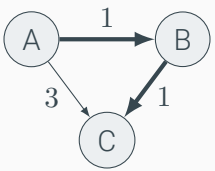
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$
  - der $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal

Vektede grafer



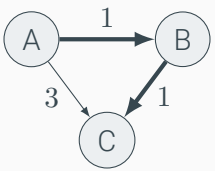
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$
  - der $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal
- Det vil si den stien som har lavest *akkumulert* vekt

Vektede grafer



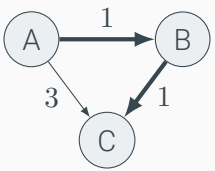
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$
  - der $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal
- Det vil si den stien som har lavest *akkumulert* vekt
- Vi skal se på to algoritmer for å finne korteste stier i vektede grafer:

Vektede grafer



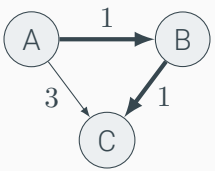
- Bredde-først søk fungerer ikke på *vektede* grafer
- En vektet graf $G = (V, E)$ har en assosiert *vektfunksjon* w
  - For en kant fra u til v i grafen, angir $w(u, v)$ vekten på kanten
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$
  - der $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal
- Det vil si den stien 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

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
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$
  - der $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal
- Det vil si den stien 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

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
- Den *korteste stien* mellom $s \in V$ og $t \in V$ er stien $v_1, v_2, \ldots, v_n$
  - der $v_1 = s$ og $v_n = t$ og $\sum_{i=1}^{n-1} w(v_i, v_{i+1})$ er minimal
- Det vil si den stien 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*

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

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*

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*
  - Det er prioritetskøen som sørger for at vi tar høyde for kantenes vekt

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*
  - Det er prioritetskøen som sørger for at vi tar høyde for kantenes vekt
  - Vi besøker alltid «nærmeste» ubesøkte noden fra startnoden med hensyn til akkumulert vekt

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*
  - Det er prioritetskøen som sørger for at vi tar høyde for kantenes vekt
  - Vi besøker alltid «nærmeste» ubesøkte noden fra startnoden med hensyn til akkumulert vekt
- Prioritetskøen ordnes etter «korteste avstand vi har sett så langt»

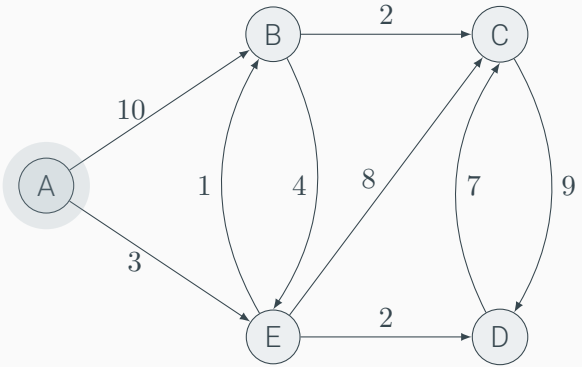
Dijkstras algoritme for korteste stier

- 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*
  - Det er prioritetskøen som sørger for at vi tar høyde for kantenes vekt
  - Vi besøker alltid «nærmeste» ubesøkte noden fra startnoden med hensyn til akkumulert vekt
- Prioritetskøen ordnes etter «korteste avstand vi har sett så langt»
- Initielt har startnoden *s* har avstand 0, og alle andre noder har avstand ∞

Dijkstras algoritme for korteste stier

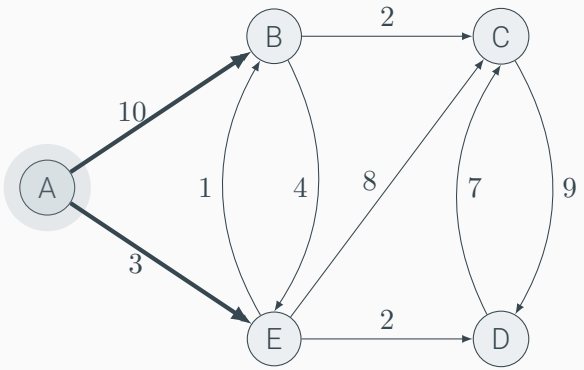
- Input er en graf *G* og en startnode *s*
- Idéen er å modifisere bredde-først søk til å ta høyde for kantenes vekt
  - Vi traverserer grafen fra *s* med en *prioritetskø*
  - Det er prioritetskøen som sørger for at vi tar høyde for kantenes vekt
  - Vi besøker alltid «nærmeste» ubesøkte noden fra startnoden med hensyn til akkumulert vekt
- Prioritetskøen ordnes etter «korteste avstand vi har sett så langt»
- Initielt har startnoden *s* har avstand 0, og alle andre noder har avstand ∞
- En node *u* legger en nabo *v* på prioritetskøen

Dijkstras algoritme for korteste stier (eksempel)



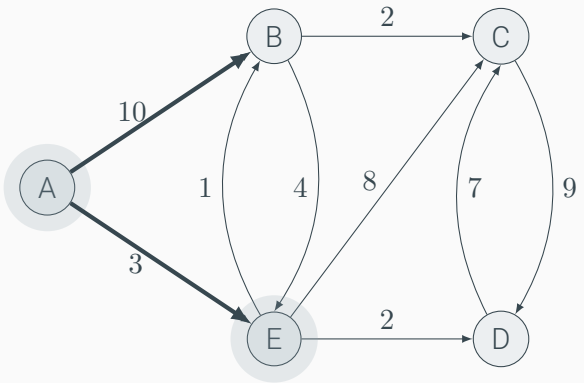
queue	priority
A	0

Dijkstras algoritme for korteste stier (eksempel)



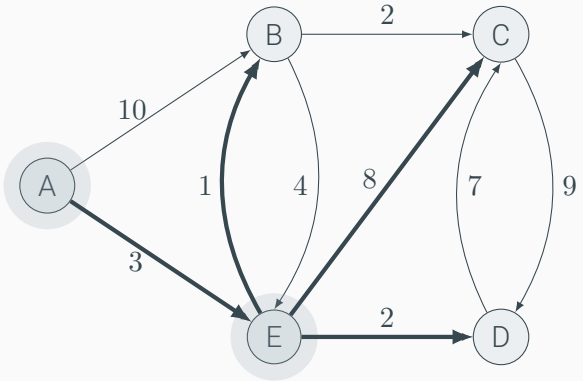
queue	priority
A	0
B	10
E	3

Dijkstras algoritme for korteste stier (eksempel)



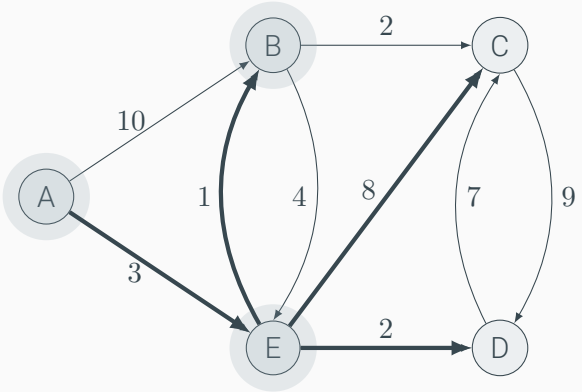
queue	priority
A	0
B	10
E	3

Dijkstras algoritme for korteste stier (eksempel)



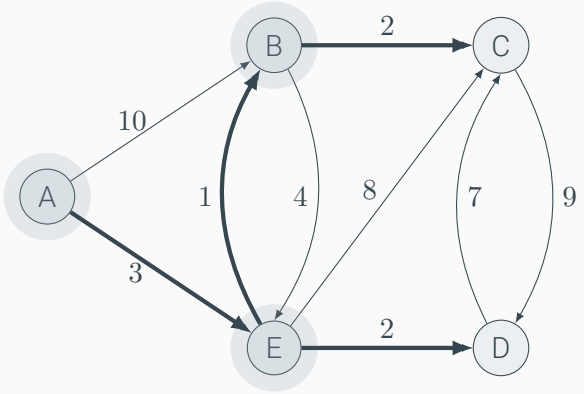
queue	priority
A	0
B	~~10~~ 4
E	3
D	5
C	11

Dijkstras algoritme for korteste stier (eksempel)



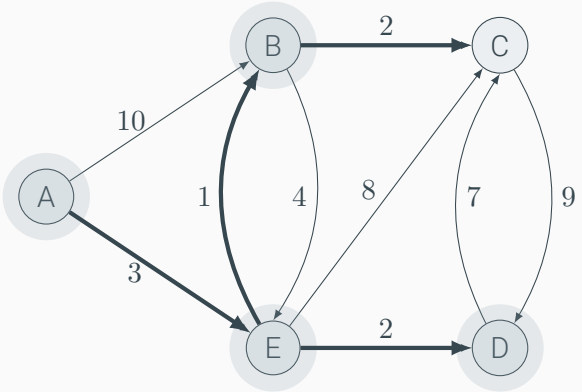
queue	priority
A	0
B	~~10~~ 4
E	3
D	5
C	11

Dijkstras algoritme for korteste stier (eksempel)



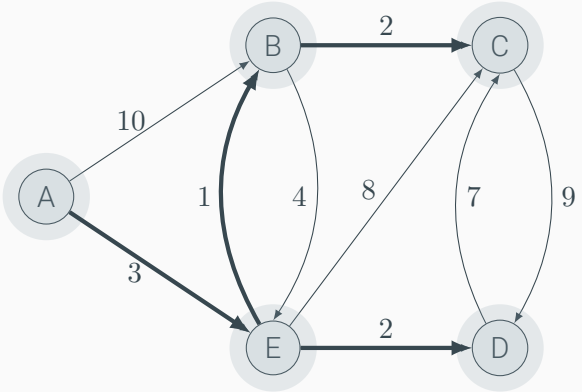
queue	priority
A	0
B	~~10~~ 4
E	3
D	5
C	~~11~~ 6

Dijkstras algoritme for korteste stier (eksempel)



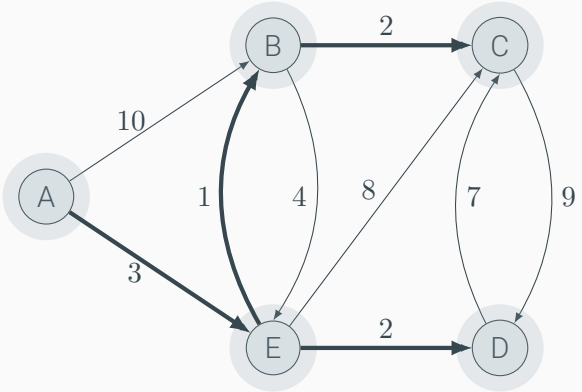
queue	priority
A	0
B	~~10~~ 4
E	3
D	5
C	~~11~~ 6

Dijkstras algoritme for korteste stier (eksempel)



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

Dijkstras algoritme for korteste stier (eksempel)



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

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

```
1 Procedure Dijkstra(G, s)
2     queue ← empty priority queue
3     dist ← empty map
```

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

```
1 Procedure Dijkstra(G, s)
2     queue ← empty priority queue
3     dist ← empty map
4     for v ∈ V do
5         dist[v] ← ∞
6         Insert(queue, v) with priority ∞
7     dist[s] ← 0
8     DecreasePriority(queue, s, 0)
```

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

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
```

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

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
```

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

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
```

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

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (kjøretidsanalyse)

- Anta at `DecreasePriority` er logaritmisk
- Hver node poppes av prioritetskøen én gang

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     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

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     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

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     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

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     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

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     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

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     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`

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     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

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     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 $\mathcal{O}((|V| + |E|) \log(|V|))$

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     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 $\mathcal{O}((|V| + |E|) \log(|V|))$
- Kan forenkles til $\mathcal{O}(|E| \cdot \log(|V|))$ hvis grafen er sammenhengende

```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

Dijkstra (implementasjon uten `DecreasePriority`)

Algorithm: Dijkstras 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

```
1 Procedure Dijkstra(G, s)
2     visited ← empty set
3     dist ← empty map with ∞ as default
4     queue ← singleton priority queue containing s with priority 0
5     dist[s] ← 0
```

Dijkstra (implementasjon uten `DecreasePriority`)

Algorithm: Dijkstras 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

```
1 Procedure Dijkstra(G, s)
2     visited ← empty set
3     dist ← empty map with ∞ as default
4     queue ← singleton priority queue containing s with priority 0
5     dist[s] ← 0
6
7     while queue is not empty and |visited| < |V| do
8         u ← RemoveMin(queue)
```

Dijkstra (implementasjon uten `DecreasePriority`)

Algorithm: Dijkstras 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

```
1  Procedure Dijkstra(G, s)
2      visited ← empty set
3      dist ← empty map with ∞ as default
4      queue ← singleton priority queue containing s with priority 0
5      dist[s] ← 0
6
7      while queue is not empty and |visited| < |V| do
8          u ← RemoveMin(queue)
9          if u ∉ visited then
10             add u to visited
```

Dijkstra (implementasjon uten `DecreasePriority`)

Algorithm: Dijkstras 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

```
1  Procedure Dijkstra(G, s)
2      visited ← empty set
3      dist ← empty map with ∞ as default
4      queue ← singleton priority queue containing s with priority 0
5      dist[s] ← 0
6
7      while queue is not empty and |visited| < |V| do
8          u ← RemoveMin(queue)
9          if u ∉ visited then
10             add u to visited
11             for (u, v) ∈ E do
12                 c ← dist[u] + w(u, v)
```

Dijkstra (implementasjon uten `DecreasePriority`)

Algorithm: Dijkstras 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

```
1  Procedure Dijkstra(G, s)
2      visited ← empty set
3      dist ← empty map with ∞ as default
4      queue ← singleton priority queue containing s with priority 0
5      dist[s] ← 0
6
7      while queue is not empty and |visited| < |V| do
8          u ← RemoveMin(queue)
9          if u ∉ visited then
10             add u to visited
11             for (u, v) ∈ E do
12                 c ← dist[u] + w(u, v)
13                 if c < dist[v] then
14                     dist[v] ← c
15                     Insert(queue, v) with priority c
```

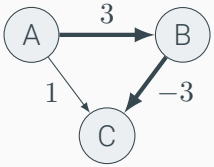
Dijkstra (implementasjon uten `DecreasePriority`)

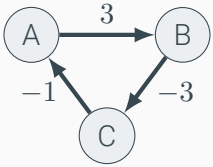
Algorithm: Dijkstras algoritme for korteste stier

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

```
1  Procedure Dijkstra(G, s)
2      visited ← empty set
3      dist ← empty map with ∞ as default
4      queue ← singleton priority queue containing s with priority 0
5      dist[s] ← 0
6
7      while queue is not empty and |visited| < |V| do
8          u ← RemoveMin(queue)
9          if u ∉ visited then
10             add u to visited
11             for (u, v) ∈ E do
12                 c ← dist[u] + w(u, v)
13                 if c < dist[v] then
14                     dist[v] ← c
15                     Insert(queue, v) with priority c
16     return dist
```

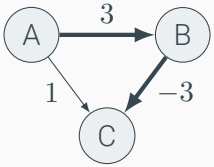
Negative vekter

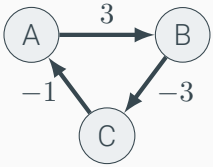




- Dijkstra kan gi feil svar for grafer med negative vekter fordi den er «grådig»

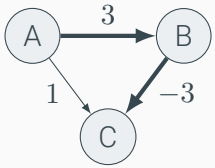
Negative vekter

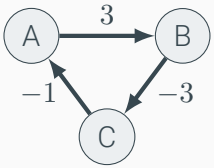




- Dijkstra kan gi feil svar for grafer med negative vekter fordi den er «grådig»
  - En grådig algoritme går ut i fra at den første løsningen er det beste

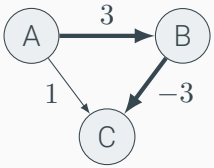
Negative vekter

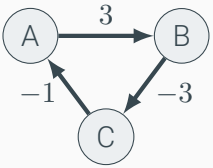




- Dijkstra kan gi feil svar for grafer med negative vekter fordi den er «grådig»
  - En grådig algoritme går ut i fra at den første løsningen er det beste
- Dersom grafen inneholder negative sykler finnes det ingen korteste sti

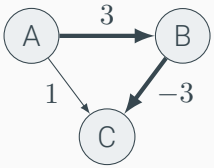
Negative vekter

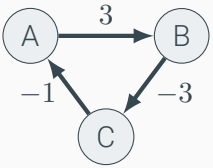




- Dijkstra kan gi feil svar for grafer med negative vekter fordi den er «grådig»
  - En grådig algoritme går ut i fra at den første løsningen er det beste
- Dersom grafen inneholder negative sykler finnes det ingen korteste sti
  - Fordi det alltid lønner seg å ta en runde til i en negativ sykel

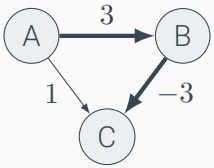
Negative vekter

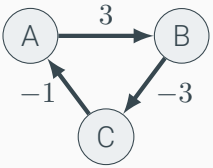




- Dijkstra kan gi feil svar for grafer med negative vekter fordi den er «grådig»
  - En grådig algoritme går ut i fra at den første løsningen er det beste
- Dersom grafen inneholder negative sykler finnes det ingen korteste sti
  - Fordi det alltid lønner seg å ta en runde til i en negativ sykel
- Bellman-Ford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

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

Bellman-Ford

- En sti kan ikke inneholde mer enn $|V| - 1$ kanter
  - En vei med $|V|$ kanter må inneholde en sykel

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

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!

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

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

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

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

```
1 Procedure BellmanFord(G, s)
2     dist ← empty map with ∞ as default
3     dist[s] = 0
4
5     repeat |V| − 1 times
6         for (u, v) ∈ E do
7             c ← dist[u] + w(u, v)
```

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

```
1 Procedure BellmanFord(G, s)
2     dist ← empty map with ∞ as default
3     dist[s] = 0
4
5     repeat |V| − 1 times
6         for (u, v) ∈ E do
7             c ← dist[u] + w(u, v)
8             if c < dist[v] then
9                 dist[v] ← c
```

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

```
1  Procedure BellmanFord(G, s)
2      dist ← empty map with ∞ as default
3      dist[s] = 0
4
5      repeat |V| − 1 times
6          for (u, v) ∈ E do
7              c ← dist[u] + w(u, v)
8              if c < dist[v] then
9                  dist[v] ← c
10
11     for (u, v) ∈ E do
12         c ← dist[u] + w(u, v)
```

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

```
1  Procedure BellmanFord(G, s)
2      dist ← empty map with ∞ as default
3      dist[s] = 0
4
5      repeat |V| − 1 times
6          for (u, v) ∈ E do
7              c ← dist[u] + w(u, v)
8              if c < dist[v] then
9                  dist[v] ← c
10
11     for (u, v) ∈ E do
12         c ← dist[u] + w(u, v)
13         if c < dist[v] then
14             error G contains a negative cycle
15     return dist
```

- Denne går gjennom alle kanter $|V|$ ganger

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

```
1  Procedure BellmanFord(G, s)
2      dist ← empty map with ∞ as default
3      dist[s] = 0
4
5      repeat |V| − 1 times
6          for (u, v) ∈ E do
7              c ← dist[u] + w(u, v)
8              if c < dist[v] then
9                  dist[v] ← c
10
11     for (u, v) ∈ E do
12         c ← dist[u] + w(u, v)
13         if c < dist[v] then
14             error G contains a negative cycle
15     return dist
```

- Denne går gjennom alle kanter $|V|$ ganger
- Det gir $\mathcal{O}(|V| \cdot |E|)$

Korteste stier i DAGs

- Hvis G er en rettet asyklisk graf (DAG), kan vi finne korteste stier i $\mathcal{O}(|V| + |E|)$

Korteste stier i DAGs

- Hvis G er en rettet asyklisk graf (DAG), kan vi finne korteste stier i $\mathcal{O}(|V| + |E|)$
- Dette gjør vi ved hjelp av en topologisk sortering

Korteste stier i DAGs

- Hvis G er en rettet asyklisk graf (DAG), kan vi finne korteste stier i $\mathcal{O}(|V| + |E|)$
- Dette gjør vi ved hjelp av en topologisk sortering

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

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

Korteste stier i DAGs

- Hvis G er en rettet asyklisk graf (DAG), kan vi finne korteste stier i $\mathcal{O}(|V| + |E|)$
- Dette gjør vi ved hjelp av en topologisk sortering

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

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

Korteste stier i DAGs

- Hvis G er en rettet asyklisk graf (DAG), kan vi finne korteste stier i $\mathcal{O}(|V| + |E|)$
- Dette gjør vi ved hjelp av en topologisk sortering

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

```
1  Procedure DAGShortestPaths(G, s)
2      dist ← empty map with ∞ as default
3      dist[s] = 0
4
5      for u ∈ TopSort(G) do
6          for (u, v) ∈ E do
7              c ← dist[u] + w(u, v)
8              if c < dist[v] then
9                  dist[v] ← c
10     return dist
```

Minimale spenntrær

Trær

- Alle trær er grafer, men ikke alle grafer er trær

Trær

- Alle trær er grafer, men ikke alle grafer er trær
- En *sammenhengende*, *urettet* og *asyklisk* graf $G = (V, E)$ er et tre

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

Trær

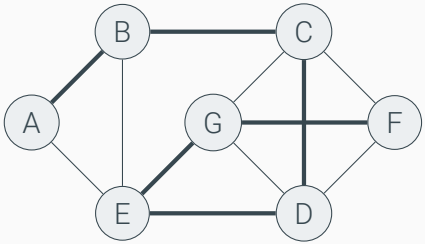
- Alle trær er grafer, men ikke alle grafer er trær
- En *sammenhengende*, *urettet* og *asyklisk* graf $G = (V, E)$ er et tre
  - Et slik tre har nøyaktig $|V| - 1$ kanter
  - Å legge til en kant i et tre vil føre til en sykel

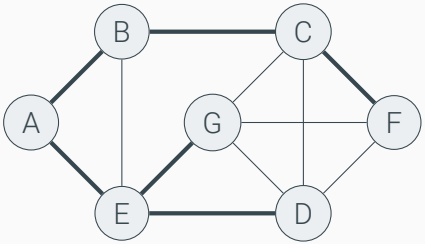
Trær

- 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

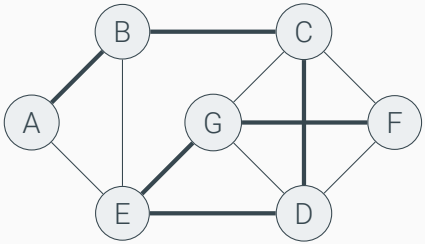
- Et *spenntre* av en sammenhengende og urettet graf $G = (V_G, E_G)$ er et tre $T = (V_T, E_T)$, der $V_T = V_G$ og $E_T \subseteq E_T$

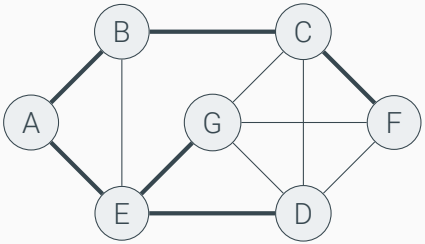




Spenntrær

- 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 $E_T \subseteq E_T$

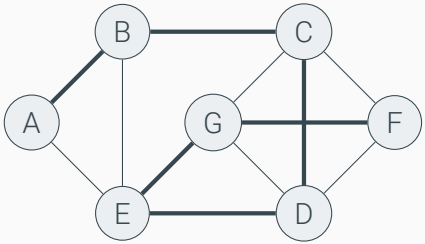


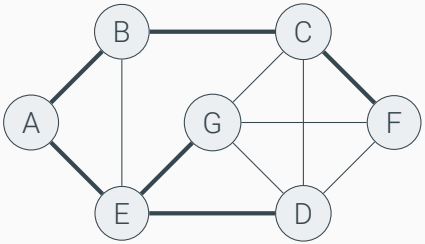


- Det vil si et tre som består av *de samme nodene* og et *utvalg* av kantene

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 $E_T \subseteq E_T$





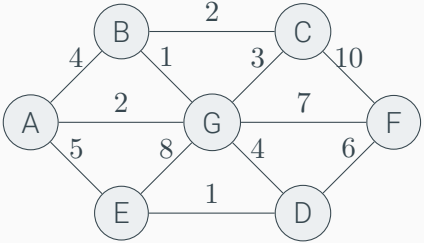
- Det vil si et tre som består av *de samme nodene* og et *utvalg* av kantene
- Vi har sett spenntrær fra bredde-først søk og Dijkstra allerede

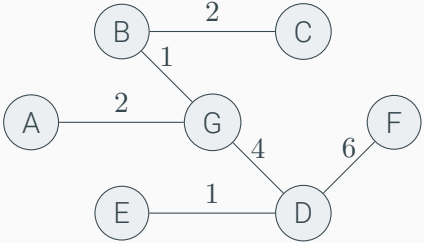
Minimale spenntrær

- Når *G* er vektet er vi ofte interessert i å finne *minimale* spenntrær

Minimale spenntrær

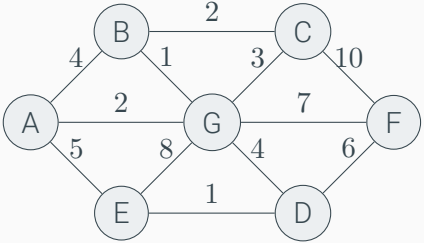
- Når *G* er vektet er vi ofte interessert i å finne *minimale* spenntrær
- Et eksempel er å koble sammen hustander med nettverkskabler billigst mulig

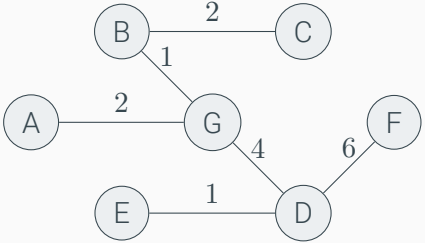




Minimale spenntrær

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

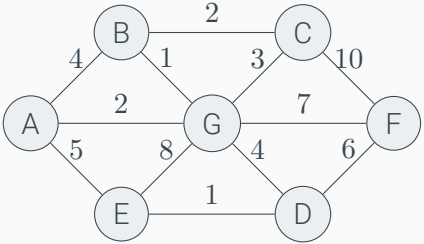


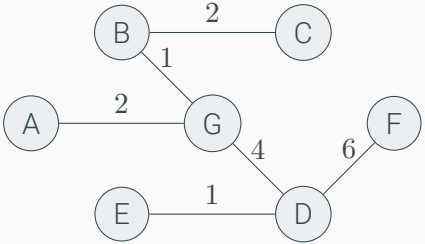


- Gitt en graf G er T et minimalt spenntre for G hvis ingen andre spenntrær for G har mindre total vekt

Minimale spenntrær

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

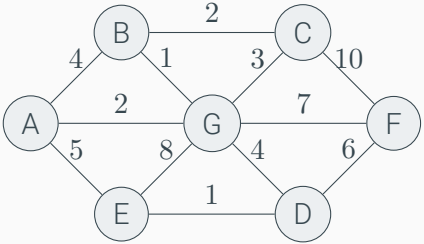


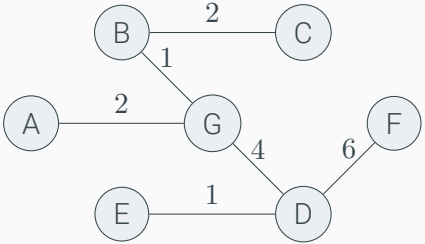


- Gitt en graf *G* er *T* et minimalt spenntre for *G* hvis ingen andre spenntrær for *G* har mindre total vekt
  - Legg merke til at det kan finnes flere minimale spenntrær for samme graf

Minimale spenntrær

- Når *G* er 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

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*

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

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*

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

Prims algoritme for minimale spenntrær

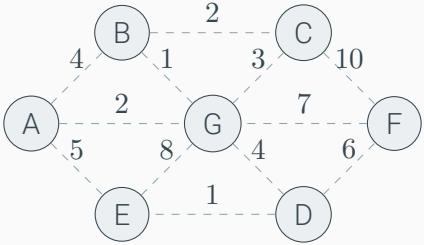
- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en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

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)

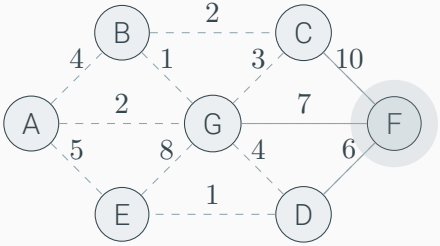
Prims algoritme for minimale spenntrær

- 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)



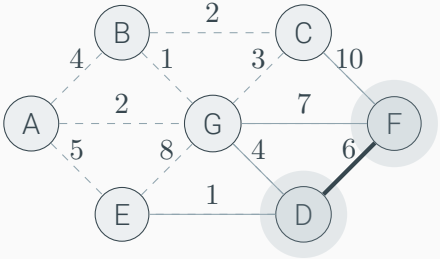
Prims algoritme for minimale spenntrær

- 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)



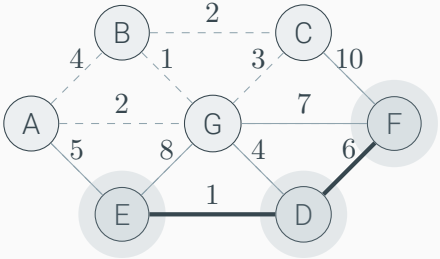
Prims algoritme for minimale spenntrær

- 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)



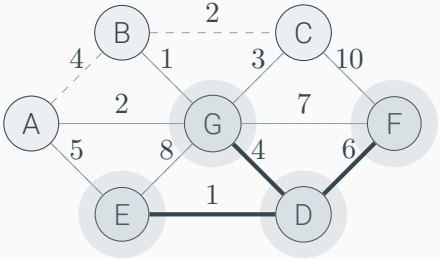
Prims algoritme for minimale spenntrær

- 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)



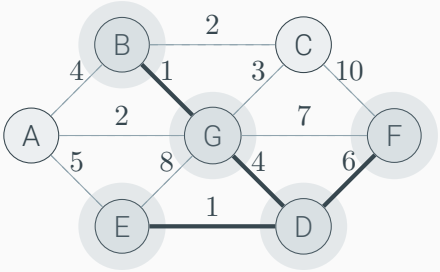
Prims algoritme for minimale spenntrær

- 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)



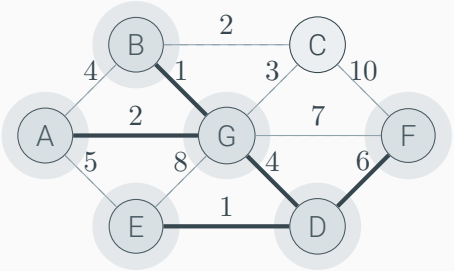
Prims algoritme for minimale spenntrær

- 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)



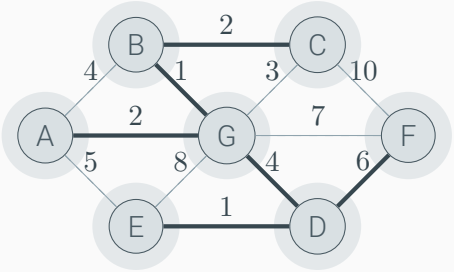
Prims algoritme for minimale spenntrær

- 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)



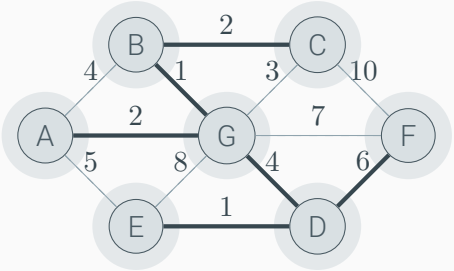
Prims algoritme for minimale spenntrær

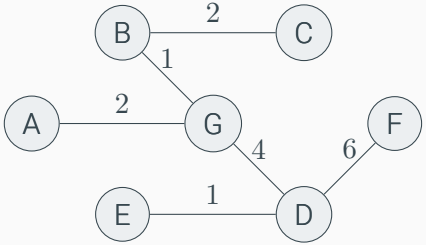
- Prims algoritme bygger opp et minimalt spenntre *T* grådig
- Vi velger vi en vilkårlig startnode til å være det uferdige spenntreet *T*
- Vi velger neste kant til å være den kanten med minst vekt
  - som sammenkobler en node fra *T* og en node som *ikke* er i *T*
- I likhet med Dijkstra bruker vi en 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

- 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

```
1 Procedure Prim(G)
2     queue ← empty priority queue
3     parents ← empty map
```

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

```
1 Procedure Prim(G)
2     queue ← empty priority queue
3     parents ← empty map
4     Insert(queue, (null, s)) with priority 0, for some arbitrary s ∈ V
5     while queue is not empty do
6         (p, u) ← RemoveMin(queue)
```

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

```
1 Procedure Prim(G)
2     queue ← empty priority queue
3     parents ← empty map
4     Insert(queue, (null, s)) with priority 0, for some arbitrary s ∈ V
5     while queue is not empty do
6         (p, u) ← RemoveMin(queue)
7         if u ∉ parents then
8             parents[u] ← p
```

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

```
1  Procedure Prim(G)
2      queue ← empty priority queue
3      parents ← empty map
4      Insert(queue, (null, s)) with priority 0, for some arbitrary s ∈ V
5      while queue is not empty do
6          (p, u) ← RemoveMin(queue)
7          if u ∉ parents then
8              parents[u] ← p
9              for (u, v) ∈ E do
10                 Insert(queue, (u, v)) with priority w(u, v)
```

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

```
1  Procedure Prim(G)
2      queue ← empty priority queue
3      parents ← empty map
4      Insert(queue, (null, s)) with priority 0, for some arbitrary s ∈ V
5      while queue is not empty do
6          (p, u) ← RemoveMin(queue)
7          if u ∉ parents then
8              parents[u] ← p
9              for (u, v) ∈ E do
10                 Insert(queue, (u, v)) with priority w(u, v)
11     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

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

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 mange flere trær (eller en mengde med trær)

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 mange flere trær (eller en mengde med trær)
- Hvis *G* er sammenhengende vil Kruskal returnere ett spenntre

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 mange 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)

Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



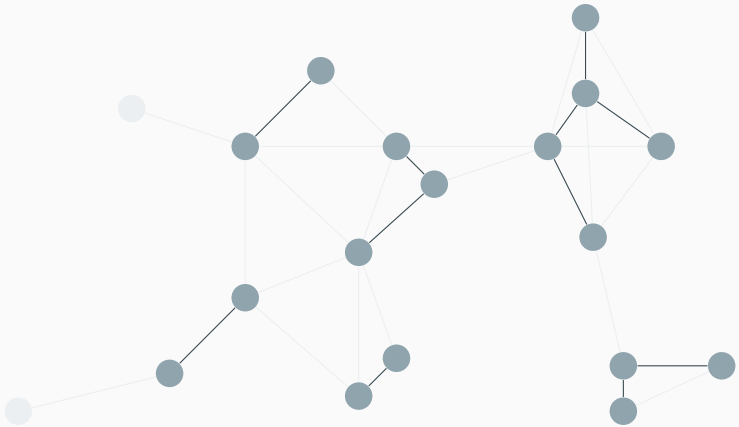
Kruskals algoritme for minimale spenntrær (illustrasjon)



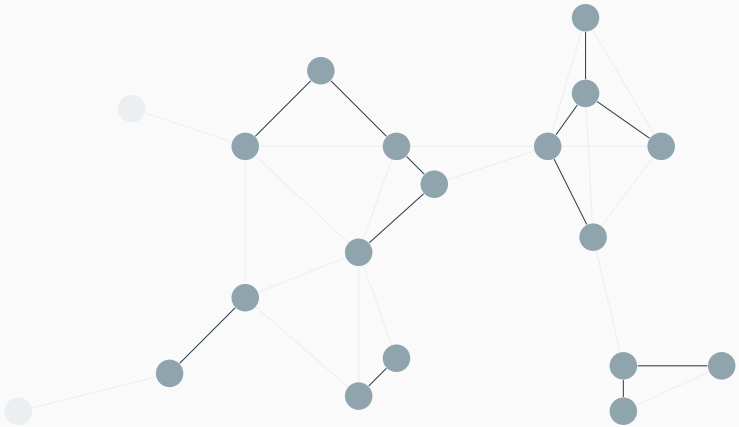
Kruskals algoritme for minimale spenntrær (illustrasjon)



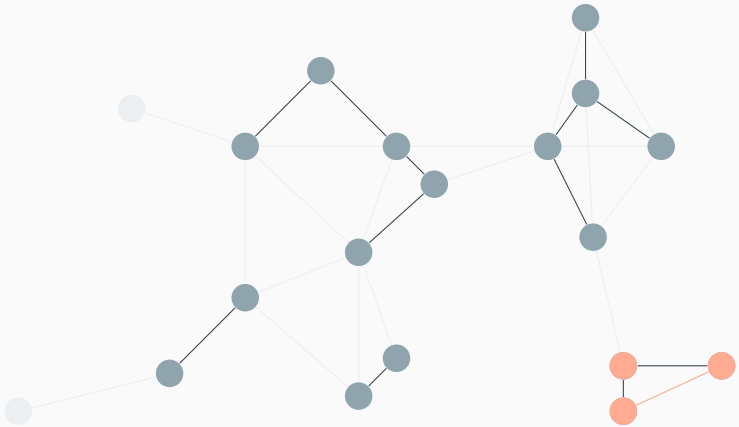
Kruskals algoritme for minimale spenntrær (illustrasjon)



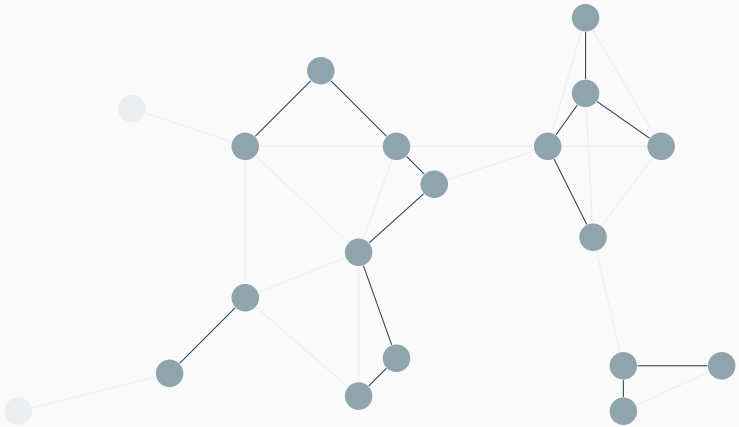
Kruskals algoritme for minimale spenntrær (illustrasjon)



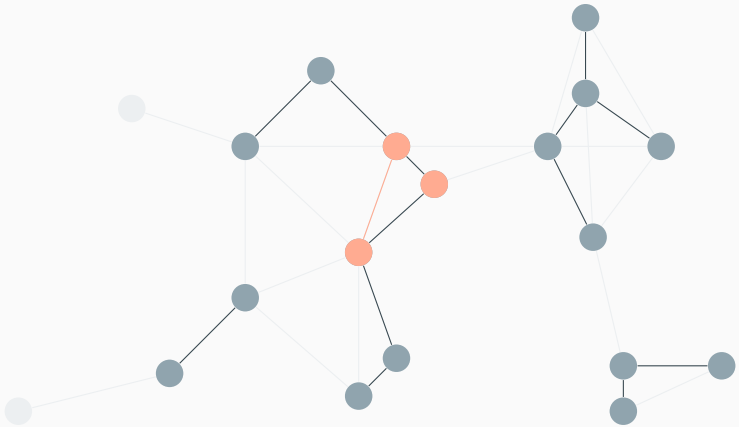
Kruskals algoritme for minimale spenntrær (illustrasjon)



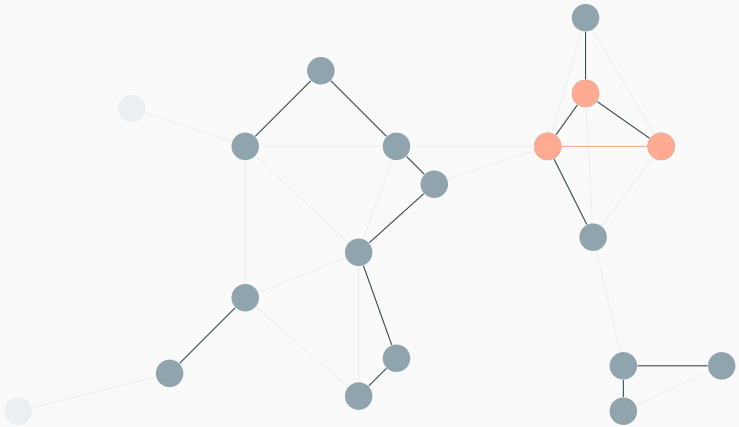
Kruskals algoritme for minimale spenntrær (illustrasjon)



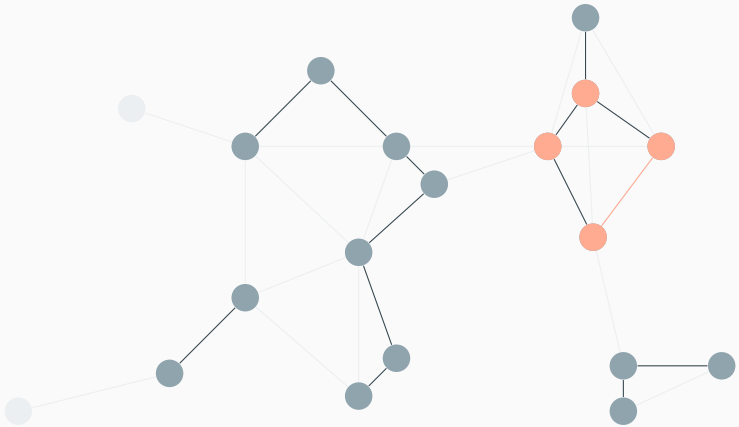
Kruskals algoritme for minimale spenntrær (illustrasjon)



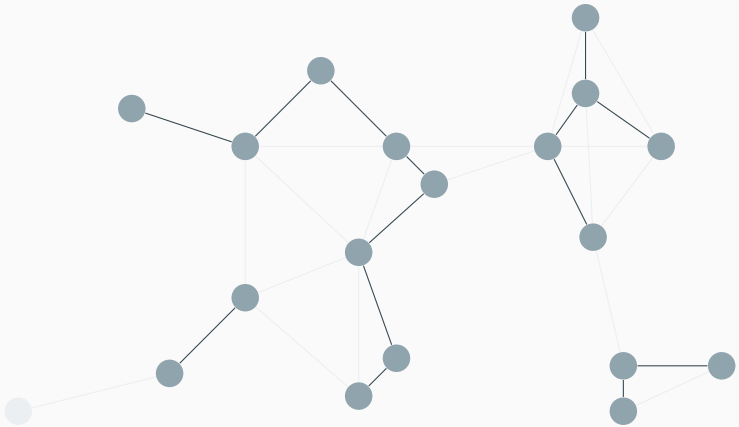
Kruskals algoritme for minimale spenntrær (illustrasjon)



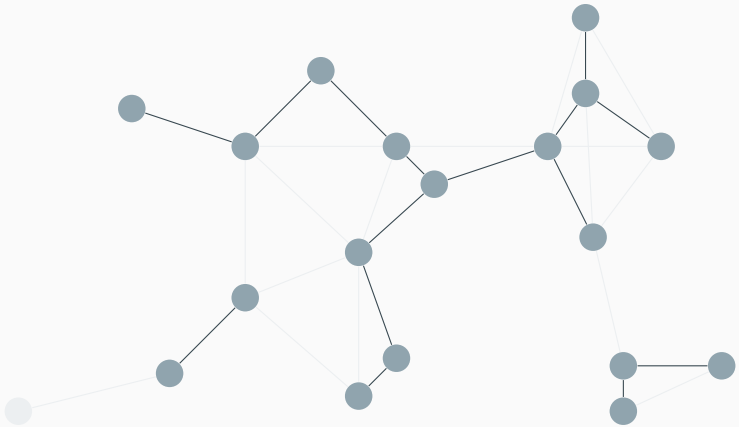
Kruskals algoritme for minimale spenntrær (illustrasjon)



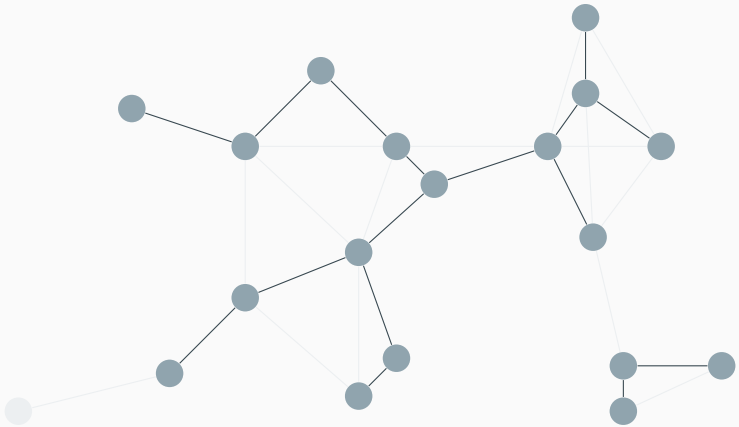
Kruskals algoritme for minimale spenntrær (illustrasjon)



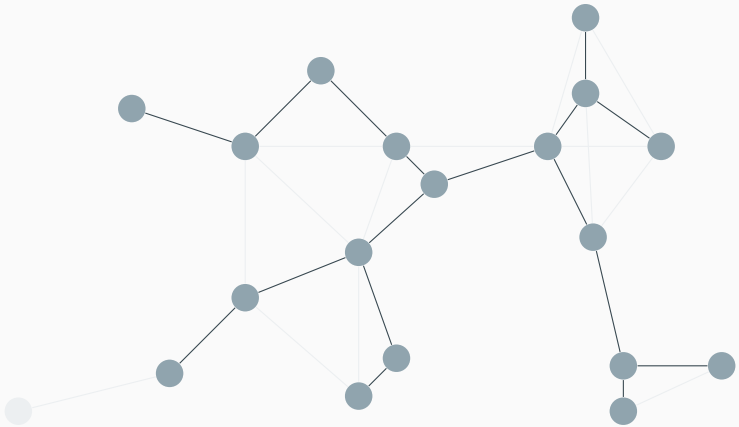
Kruskals algoritme for minimale spenntrær (illustrasjon)



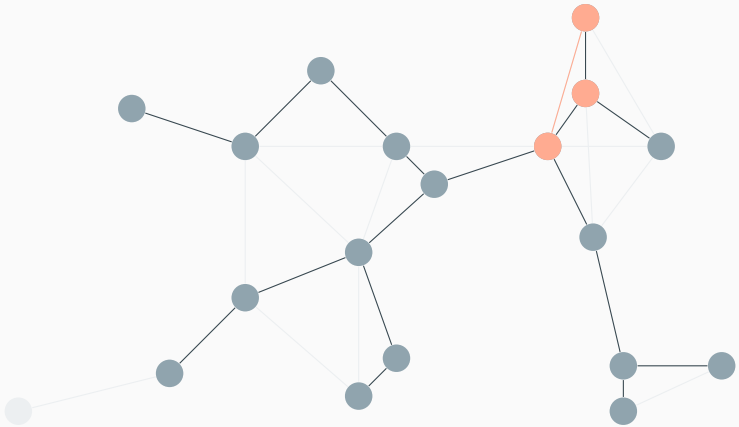
Kruskals algoritme for minimale spenntrær (illustrasjon)



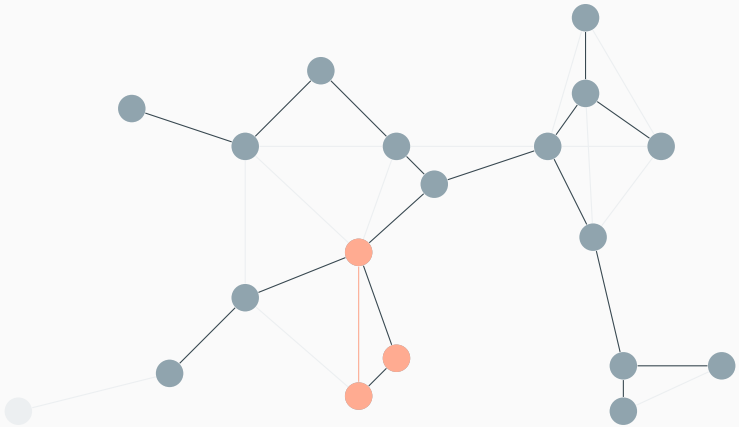
Kruskals algoritme for minimale spenntrær (illustrasjon)



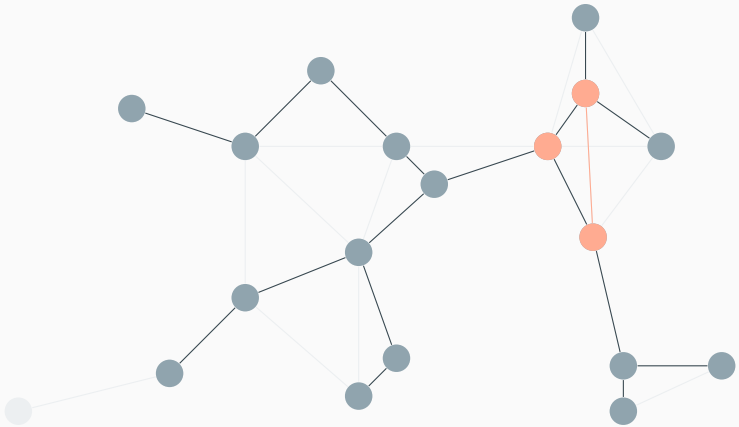
Kruskals algoritme for minimale spenntrær (illustrasjon)



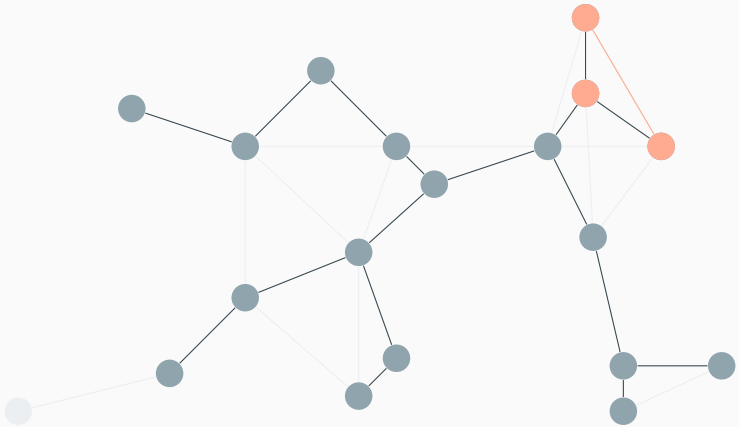
Kruskals algoritme for minimale spenntrær (illustrasjon)



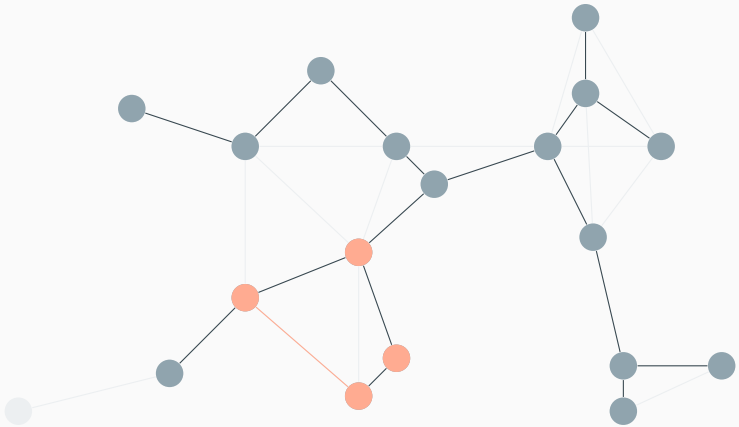
Kruskals algoritme for minimale spenntrær (illustrasjon)



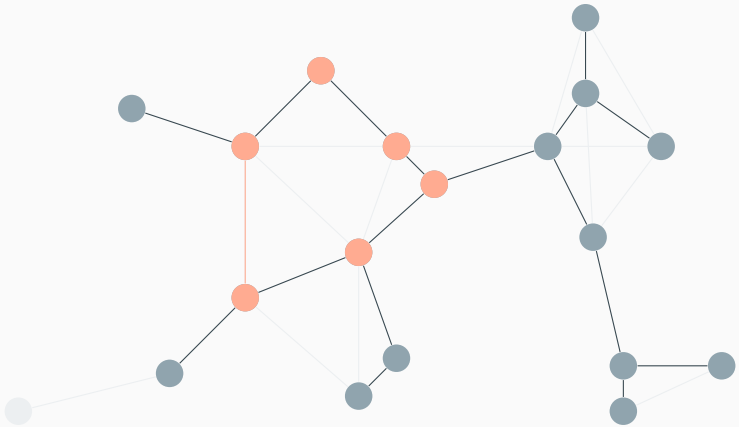
Kruskals algoritme for minimale spenntrær (illustrasjon)



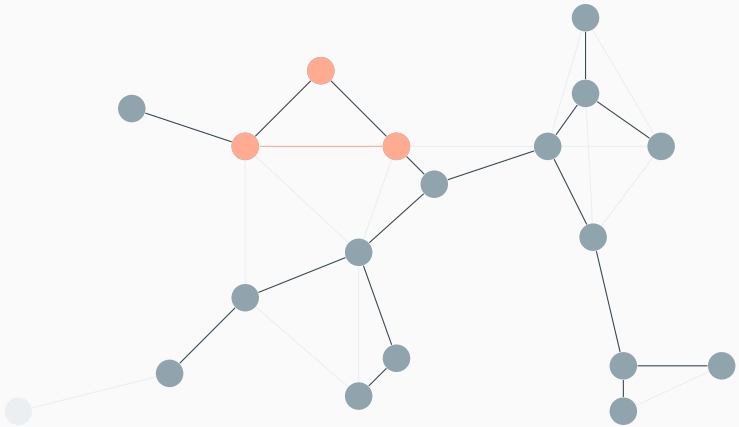
Kruskals algoritme for minimale spenntrær (illustrasjon)



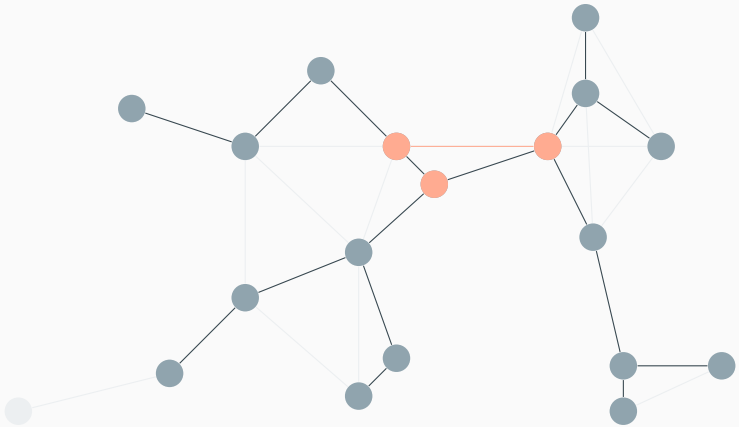
Kruskals algoritme for minimale spenntrær (illustrasjon)



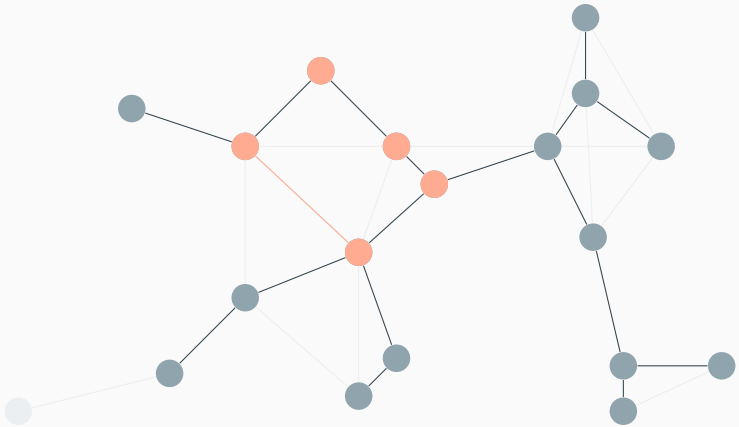
Kruskals algoritme for minimale spenntrær (illustrasjon)



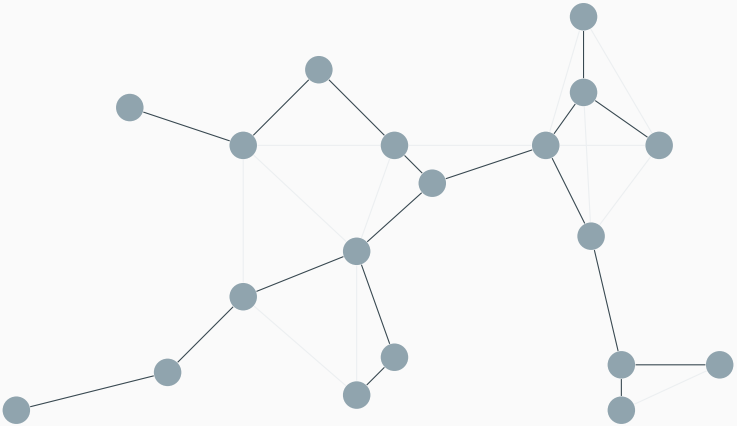
Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Kruskals algoritme for minimale spenntrær (illustrasjon)



Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas

Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas
- I likhet med Kruskal gir den et minimalt spenntre for hver komponent i grafen *G*

Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas
- I likhet med Kruskal gir den et minimalt spenntre for hver komponent i grafen *G*
- Den går ut på å anse hver node i grafen som et tre (som kun består av den noden)

Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas
- I likhet med Kruskal gir den et minimalt spenntre for hver komponent i grafen *G*
- Den går ut på å anse hver node i grafen som et tre (som kun består av den noden)
  - Disse trærene utgjør en skog

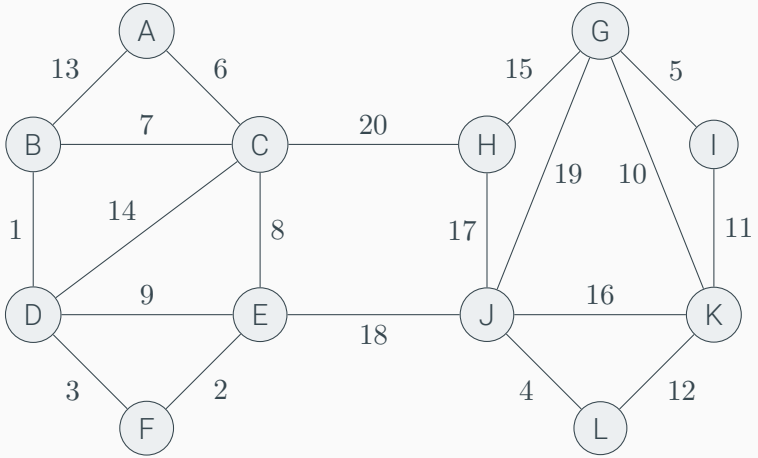
Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas
- I likhet med Kruskal gir den et minimalt spenntre for hver komponent i grafen *G*
- Den går ut på å anse hver node i grafen som et tre (som kun består av den noden)
  - Disse trærene utgjør en skog
- Iterativt gå over alle trærene velg den billigste kanten som forbinder treet med et annet

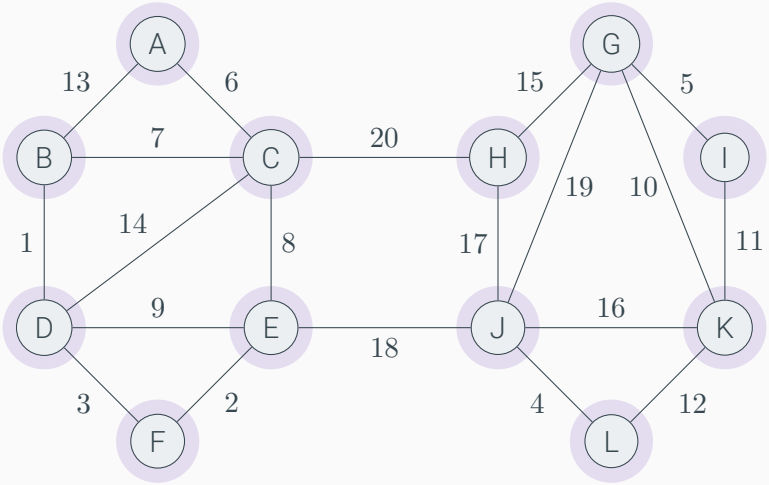
Borůvkas

- En siste algoritme for minimale spenntrær er Borůvkas
- I likhet med Kruskal gir den et minimalt spenntre for hver komponent i grafen *G*
- Den går ut på å anse hver node i grafen som et tre (som kun består av den noden)
  - Disse trærene utgjør en skog
- Iterativt gå over alle trærene velg den billigste kanten som forbinder treet med et annet
- Algoritmen terminerer når det ikke lenger finnes kanter som forbinder forskjellige trær

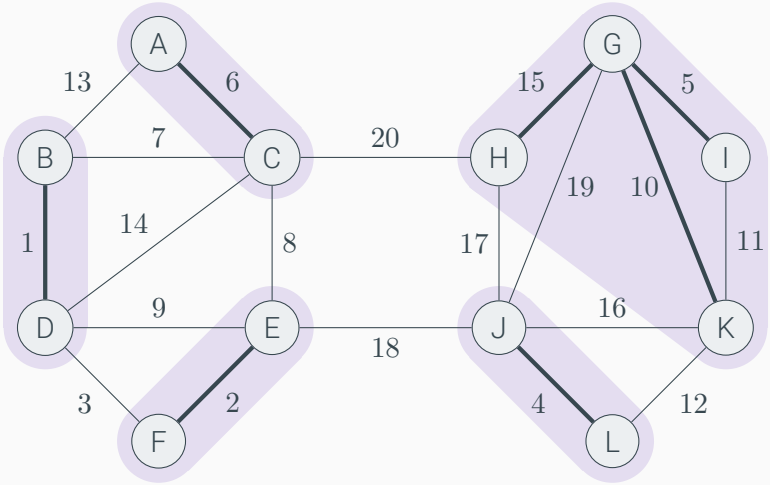
Borůvkas (eksempel)



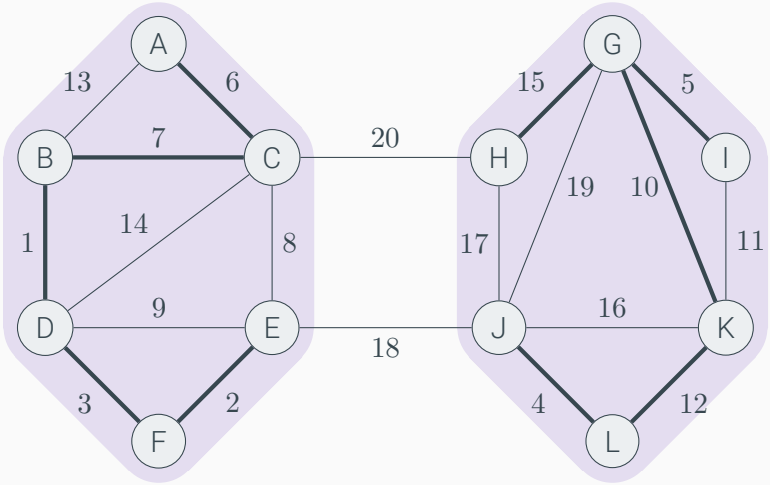
Borůvkas (eksempel)



Borůvkas (eksempel)



Borůvkas (eksempel)



Borůvkas (eksempel)

