Korteste veje

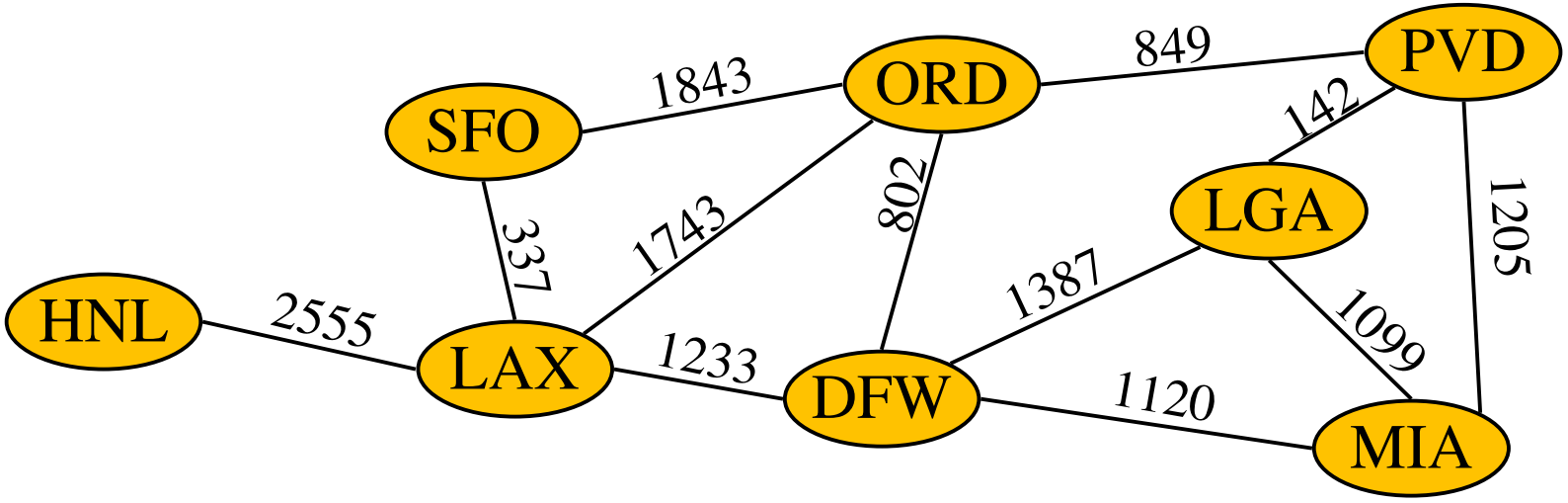
Vægtede grafer

I en **vægtet graf** har enhver kant tilknyttet en numerisk værdi, kaldet kantens **vægt**

Vægte kan repræsentere afstande, omkostninger, o.s.v.

Eksempel:

I en flyrutegraf repræsenterer vægten af en kant afstanden imellem to lufthavne



Korteste vej

Givet en vægtet graf og to knuder ***u*** og ***v*** ønsker vi at finde en vej imellem ***u*** og ***v***, der har minimal samlet vægt

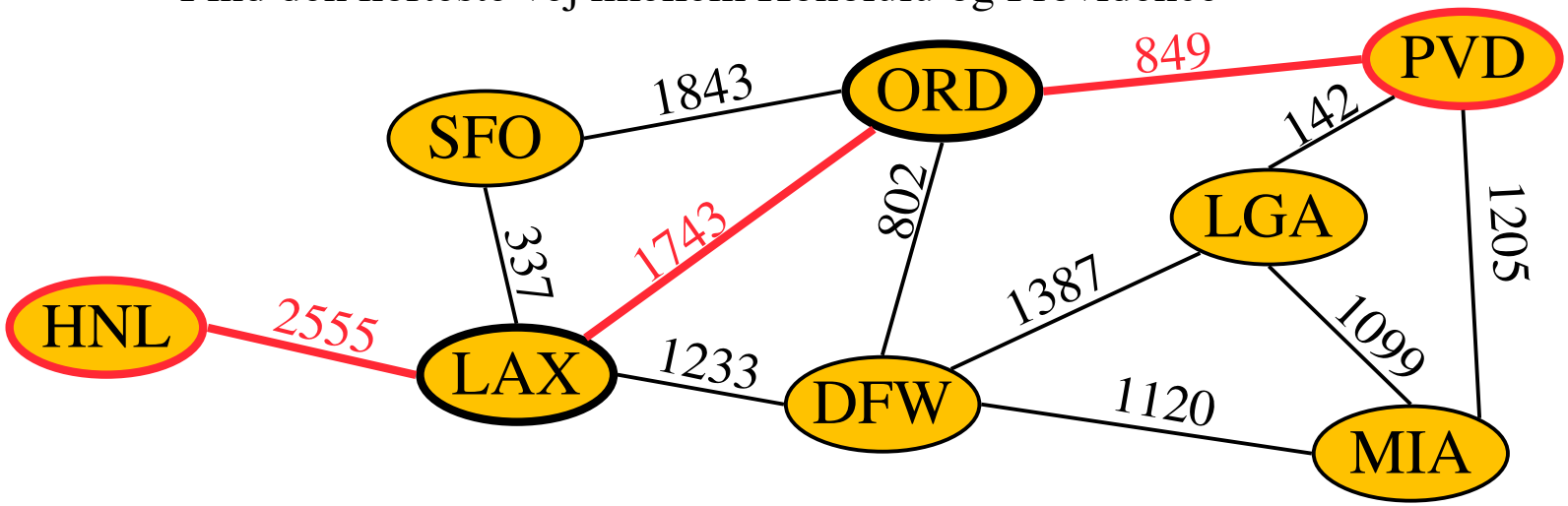
Længden af en vej er summen af dens kantvægte

Anvendelser

- Routing af internetpakker
- Ruteplanlægning (bil/fly)

Eksempel:

- Find den korteste vej imellem Honolulu og Providence



Egenskaber ved korteste vej

Egenskab 1:

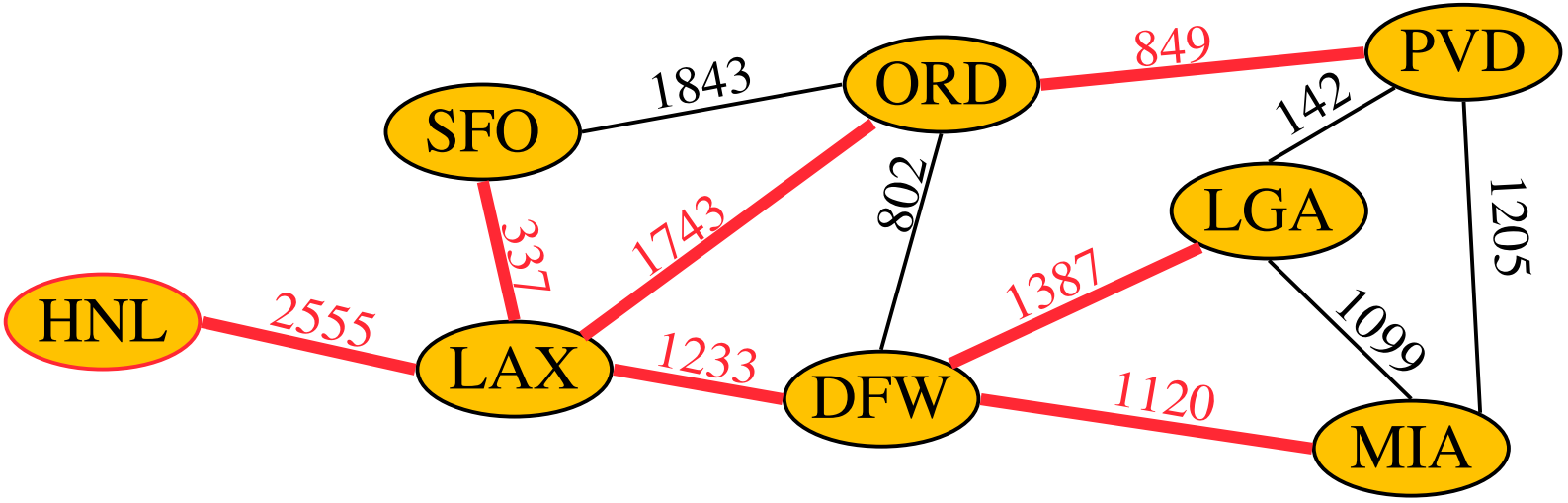
En delvej af den korteste vej er selv en korteste vej

Egenskab 2:

Der findes et træ af korteste veje fra en startknude til alle andre knuder

Eksempel:

Træ af korteste veje fra Honolulu



Dijkstra's algoritme

Afstanden for en knude ***v*** fra en knude ***s*** er længden af en korteste vej imellem ***s*** og ***v***

Dijkstra's algoritme beregner afstandene for alle knuder fra en given startknude ***s***

Forudsætninger:

- grafen er sammenhængende
- kanterne er ikke-orienterede
- kantvægtene er ikke-negative

Vi lader en "**sky**" af knuder vokse, begyndende med ***s*** og slutteligt dækkende alle knuder

Vi gemmer for hver knude ***v*** en etikette ***d***(***v***), der repræsenterer afstanden for ***v*** fra ***s*** i den delgraf, der består af skyen og dens tilstødende knuder

Vi udvider skyen med den knude, ***u***, uden for skyen, som har mindst afstand, ***d***(***u***)

Vi opdaterer etiketterne for de knuder uden for skyen, der er naboer til ***u***

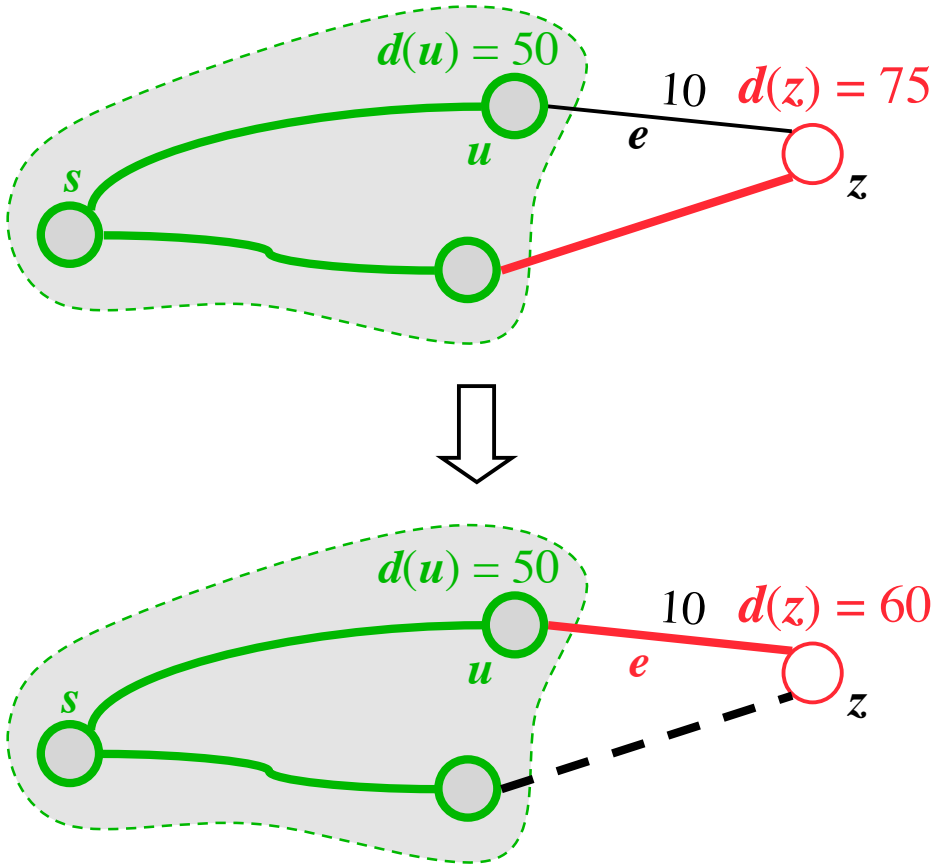
Kant-relaxation

Betragt en kant $e = (u,z)$, hvor

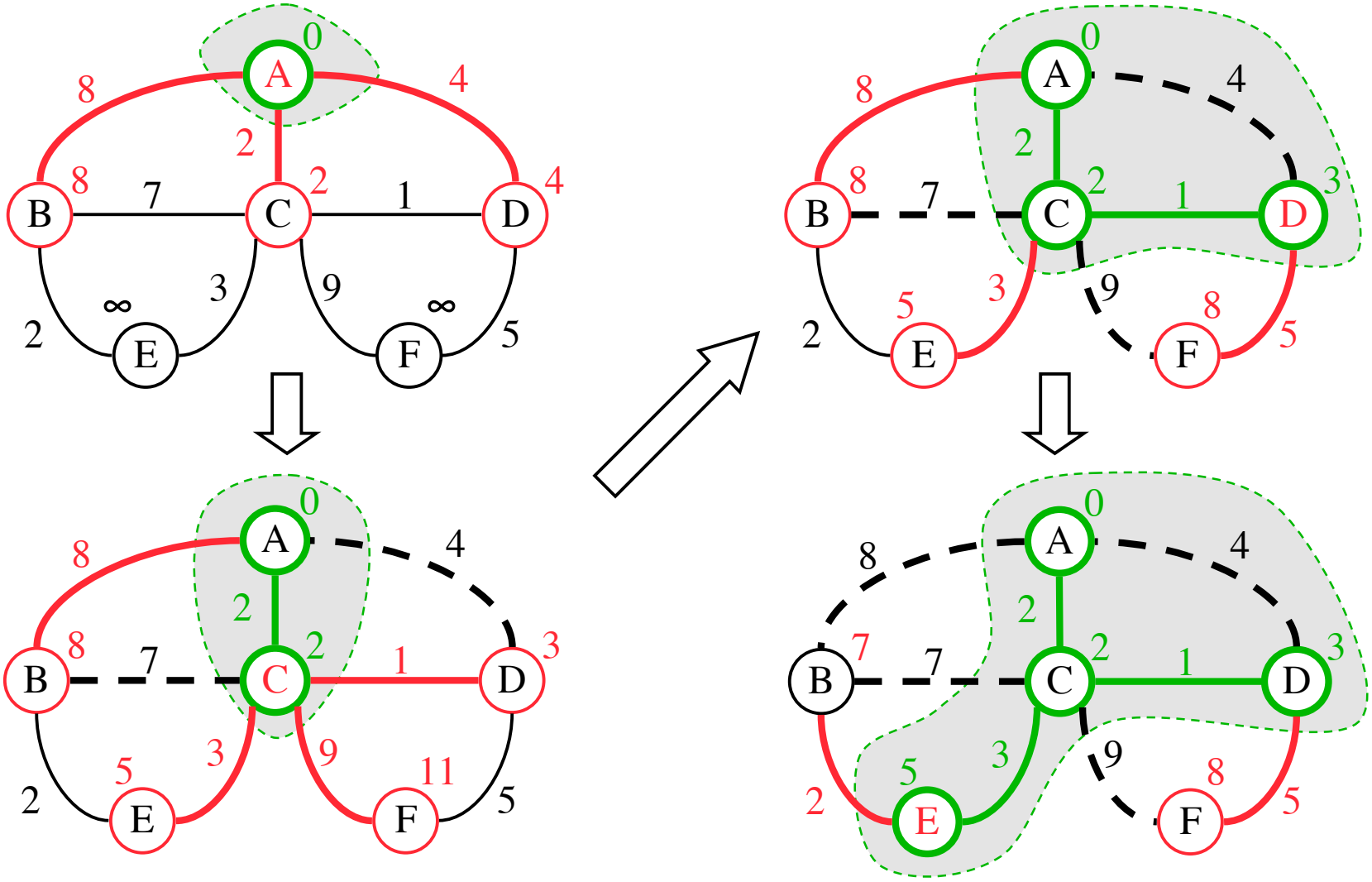
- u er den knude, der senest er blevet tilføjet skyen
- z ikke er i skyen

En relaxation med kanten e opdaterer afstanden $d(z)$, som følger:

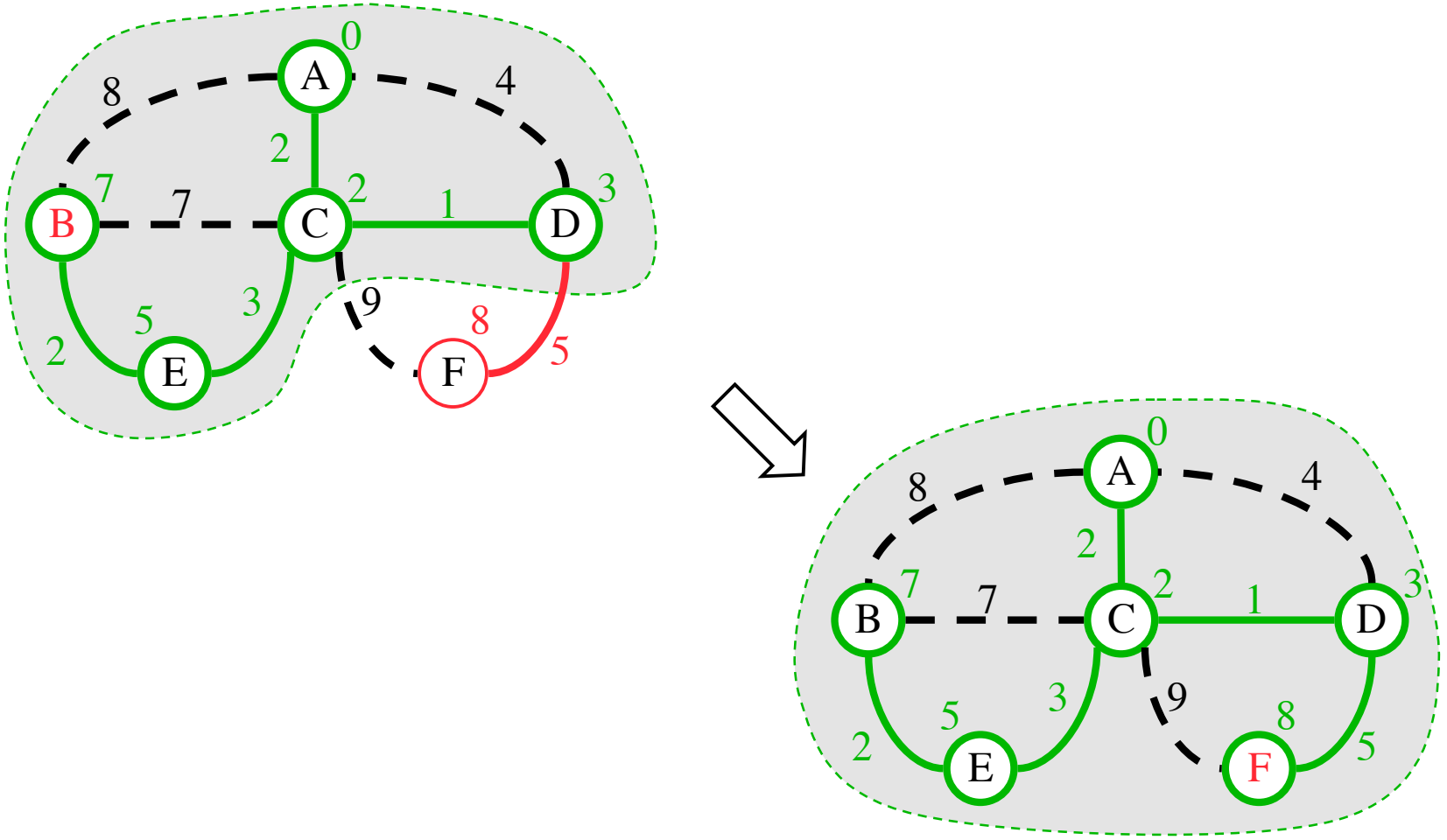
$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$



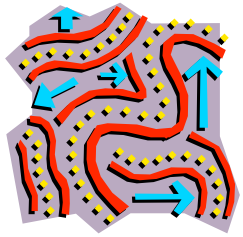
Eksempel



Eksempel (fortsat)



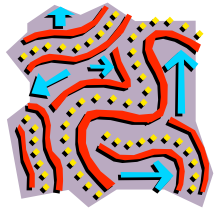
Dijkstra's algoritme



- En prioritetskø indeholder de knuder, der er uden for skyen
  - Nøgle: afstand
  - Element: knude
- Locator-baserede metoder:
  - *insert*(*k*,*e*)
    returnerer en locator
  - *replaceKey*(*l*,*k*)
    ændrer nøglen for et emne
- For hver knude ***v*** gemmes to etiketter:
  - afstand, ***d***(***v***)
  - locator i prioritetskøen

```
Algorithm DijkstraDistances(G, s)
    Q ← new heap-based priority queue
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
        l ← Q.insert(getDistance(v), v)
        setLocator(v,l)
    while ¬Q.isEmpty()
        u ← Q.removeMin()
        for all e ∈ G.incidentEdges(u)
            { relax edge e }
            z ← G.opposite(u,e)
            r ← getDistance(u) + weight(e)
            if r < getDistance(z)
                setDistance(z,r)
                Q.replaceKey(getLocator(z),r)
```

Analyse



- Grafoperationer:
  Metoden **incidentEdges** kaldes en gang for hver knude

- Etiketteoperationer:
  Vi sætter eller henter afstands- og locator-etiketterne for en knude z
  $O(\deg(z))$ gange. Hver sådan operation tager $O(1)$ tid

- Prioritetskøoperationer:
  Hver knude indsættes en gang i og fjernes en gang fra prioritetskøen
  Hver indsættelse og fjernelse tager $O(\log n)$ tid
  Nøglen for en knude w i prioritetetskøen ændres højst $\deg(w)$ gange,
  hvor hver ændring tager $O(\log n)$ tid

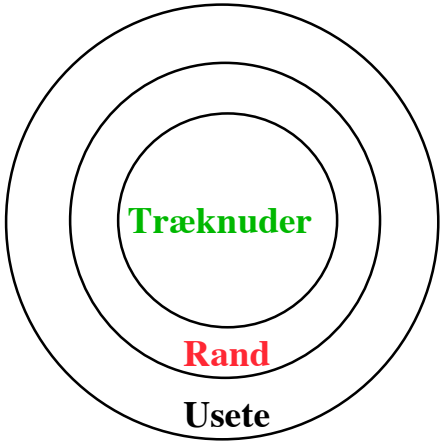
- Dijkstra's algoritme kører i $O((n + m) \log n)$ tid, hvis grafen er repræsenteret ved en nabolistestruktur (husk, at $\sum_v \deg(v) = 2m$)

- Køretiden kan også udtrykkes som $O(m \log n)$, da grafen er sammenhængende

Klassedeling af knuder

I algoritmen skelnes imellem 3 typer af knuder:

(1) **Træknuder** (knuder i skyen)
(2) **Randknuder** (knuder, der støder op til træknuder)
(3) **Usete knuder** (alle andre knuder)



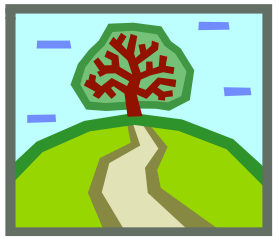
Bedste-først-søgning

(priority-first search)

Anvend en prioritetskø, *pq* (i stedet for en stak eller en kø)

Sæt alle knuders status til *UNSEEN*
Læg en knude i *pq* og sæt dens status til *FRINGE*
Så længe *pq* ikke er tom:
    Fjern den "bedste" knude, *u*, fra *pq*
    Sæt *u*'s status til *TREE*
    For enhver af *u*'s naboknuder:
        hvis *UNSEEN*, så læg den i *pq* og sæt dens status til *FRINGE*,
        hvis *FRINGE*, så opdater dens prioritet

Udvidelse



Vi kan udvide Dijkstra's algoritme til at returnere et træ af korteste veje imellem startknuden og alle andre knuder

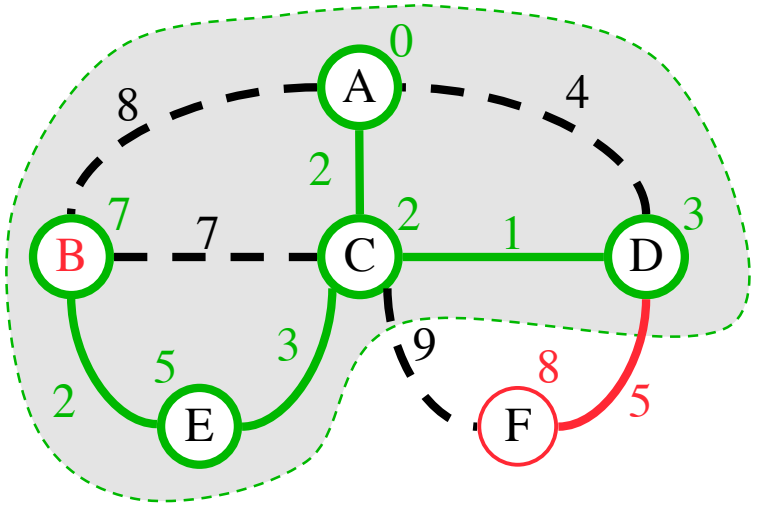
For hver knude gemmer vi en tredje etikette, der indeholder forældrekanten i det aktuelt korteste-vej-træ

```
Algorithm DijkstraShortestPathsTree(G, s)

    …

  for all v ∈ G.vertices()
    …
    setParent(v, ∅)
    …

    for all e ∈ G.incidentEdges(u)
      { relax edge e }
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
        setParent(z,e)
        Q.replaceKey(getLocator(z),r)
```

Hvorfor Dijkstra's algoritme virker

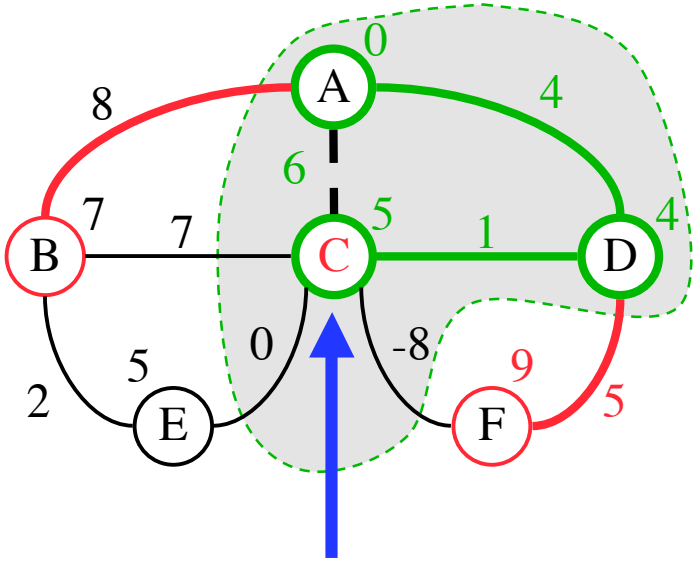
Dijkstra's algoritme er baseret på den grådige metode
Den tilføjer knuder efter voksende afstand (mindste-først)

- Antag at algoritmen ikke fandt alle korteste afstande. Lad F være den første *forkerte* knude, algoritmen behandlede
- Dengang den foregående knude, D, på den rigtige korteste vej blev betragtet, var dens afstand korrekt
- Men en relaxation med kanten (D,F) blev foretaget på dette tidspunkt!
- Derfor kan F's afstand ikke være forkert, så længe $\boldsymbol{d}(F) \geq \boldsymbol{d}(D)$
- Der kan altså ikke være nogen *forkert* knude



Hvorfor algoritmen ikke virker for negative kantvægte

Hvis en knude med en nabokant med negativ vægte skulle tilføjes sent til skyen, ville det spolere afstandene for de knuder, der allerede er i skyen



C's korrekte afstand er 1,
men C er allerede i skyen med $d(C)=5$

Bellman-Ford's algoritme

Virker selv for negative vægte

Vi må forudsætte, at kanterne er orienterede (ellers kunne vi få en cykel med negativ vægt)

Iteration ***i*** finder alle korteste veje fra *s*, der bruger ***i*** kanter

Køretid: $O(nm)$

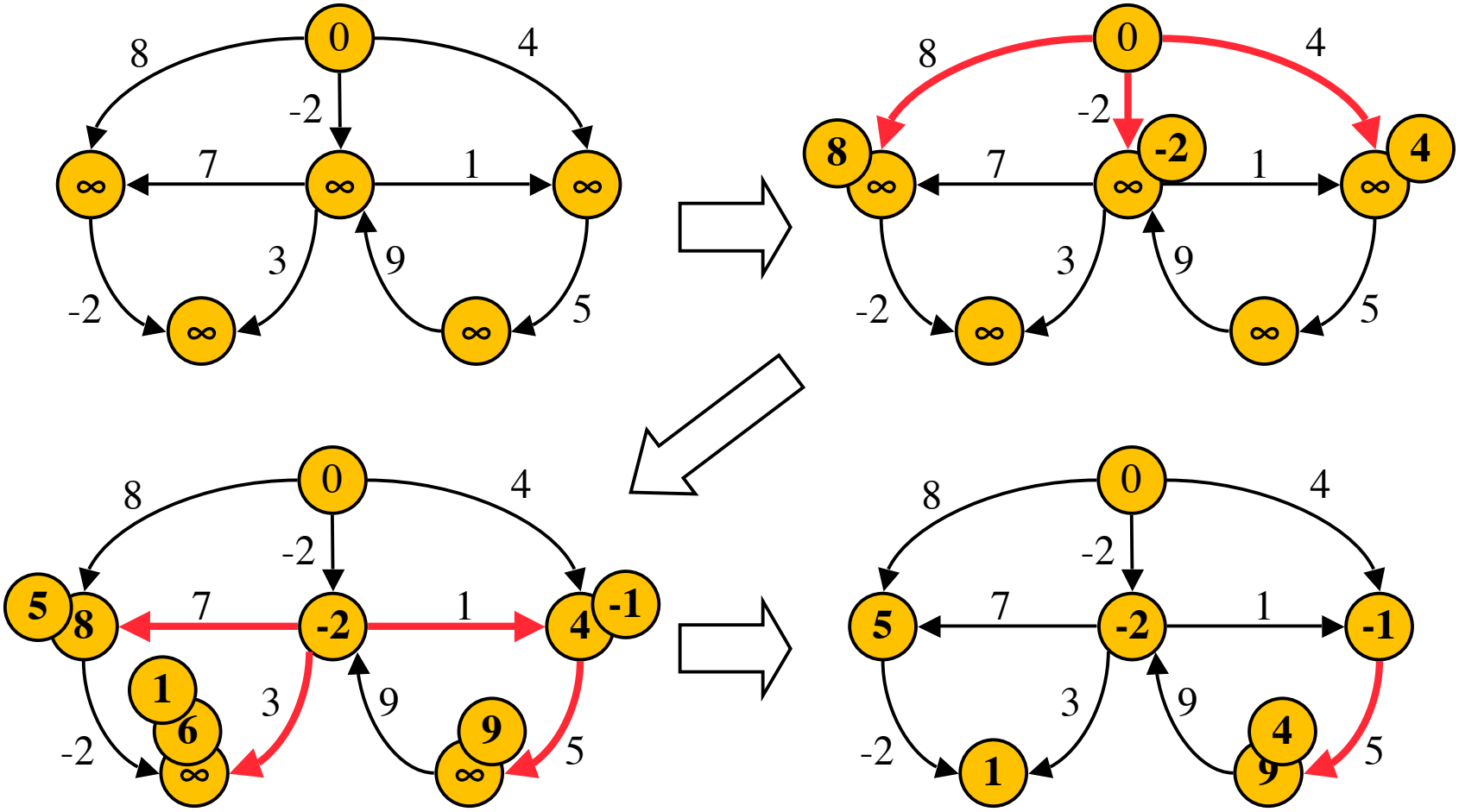
Kan udvides til at afsløre en cykel med negativ vægt, hvis en sådan findes

Hvordan?

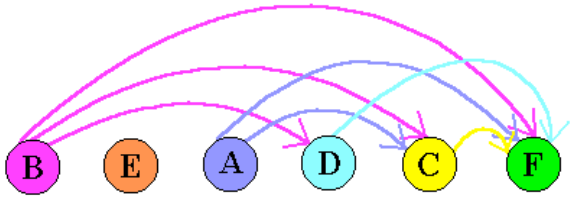
```
Algorithm BellmanFord(G, s)
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
    for i ← 1 to n-1 do
        for each e ∈ G.edges()
            { relax edge e }
            u ← G.origin(e)
            z ← G.opposite(u,e)
            r ← getDistance(u) + weight(e)
            if r < getDistance(z)
                setDistance(z,r)
```

Bellman-Ford eksempel

Knuder er mærket med deres afstand, $d(v)$



DAG-baseret algoritme



Virker selv for negative vægte

Bruger topologisk sortering

Bruger ingen specielle datastrukturer

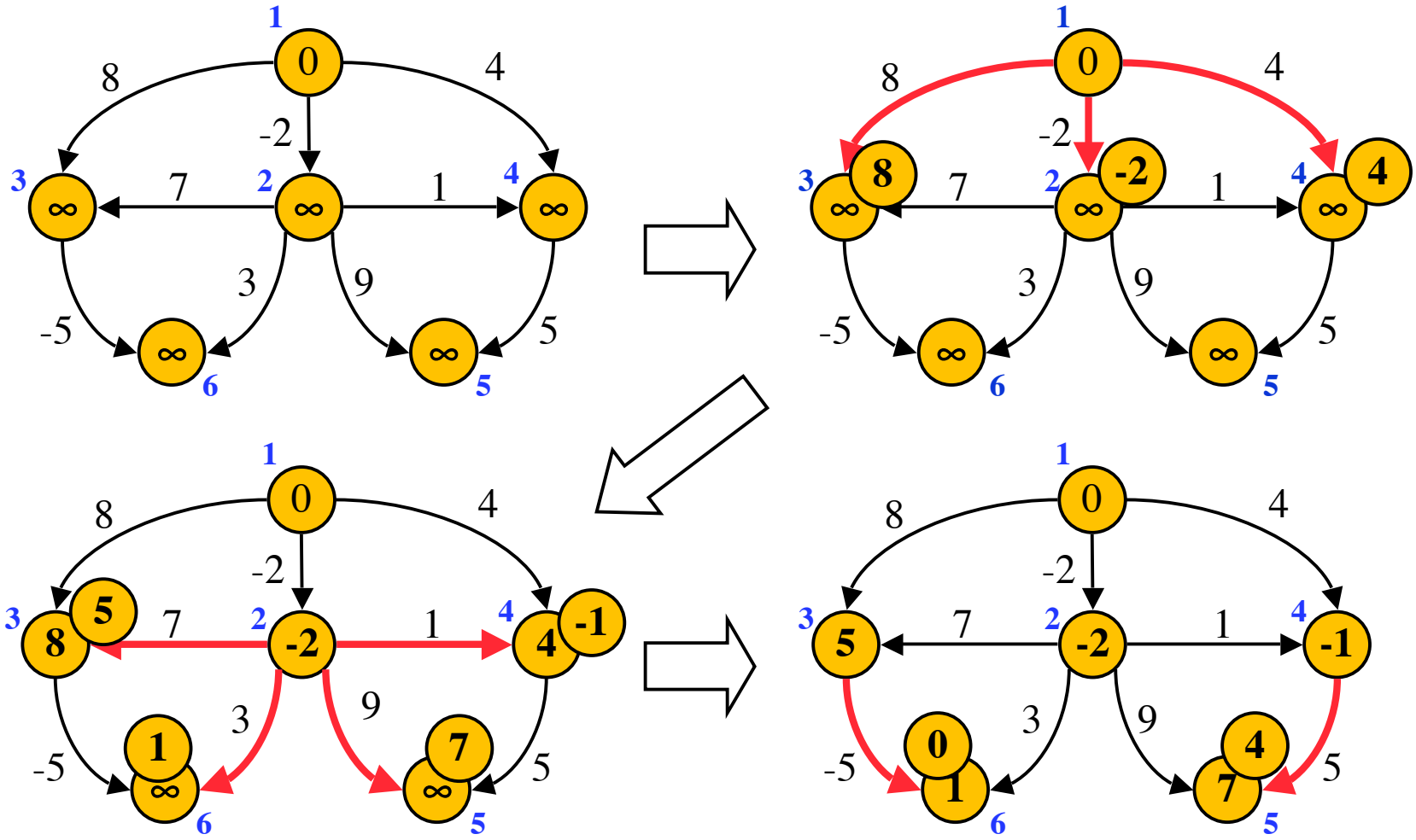
Er meget hurtigere end Dijkstra's algoritme

Køretid: $O(n+m)$

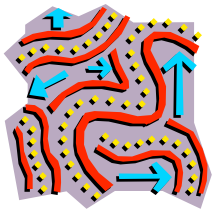
```
Algorithm DagDistances(G, s)
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  Perform a topological sort of the vertices
  for u ← 1 to n do    {in topological order}
    for each e ∈ G.outEdges(u)
      { relax edge e }
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
```

DAG eksempel

Knuder er mærket med deres afstand, $d(v)$



Alle korteste veje



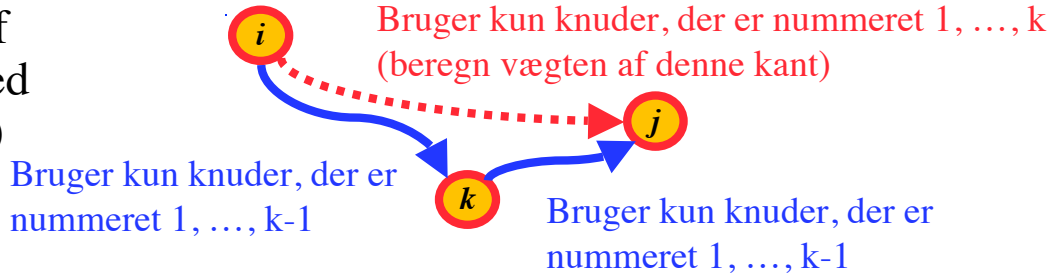
Bestem den korteste vej imellem ethvert par af knuder i en vægtet orienteret graf G

Vi kan udføre n kald af Dijkstra's algoritme (hvis der ikke er negative kanter), hvilket tager $O(nm \log n)$ tid

Tilsvarende ville n kald af Bellman-Ford's algoritme tage $O(n^2m)$ tid

Vi kan opnå $O(n^3)$ tid ved brug af dynamisk programmering (i lighed med Floyd-Warshall's algoritme)

Algorithm *AllPair(G)* {assumes vertices 1,…,*n*}
for all *vertex pairs (i,j)*
  if $i = j$
    $D_0[i,i] \leftarrow 0$
  else if *(i,j) is an edge in G*
    $D_0[i,j] \leftarrow$ *weight of edge (i,j)*
  else
    $D_0[i,j] \leftarrow +\infty$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $D_k[i,j] \leftarrow \min\{D_{k-1}[i,j], D_{k-1}[i,k]+D_{k-1}[k,j]\}$
return D_n



Mindste udspændende træer

Mindste udspændende træ

Udspændende delgraf

Delgraf af en graf ***G***, der indeholder all ***G***'s knuder

Udspændende træ

Udspændende delgraf, der et et (frit) træ

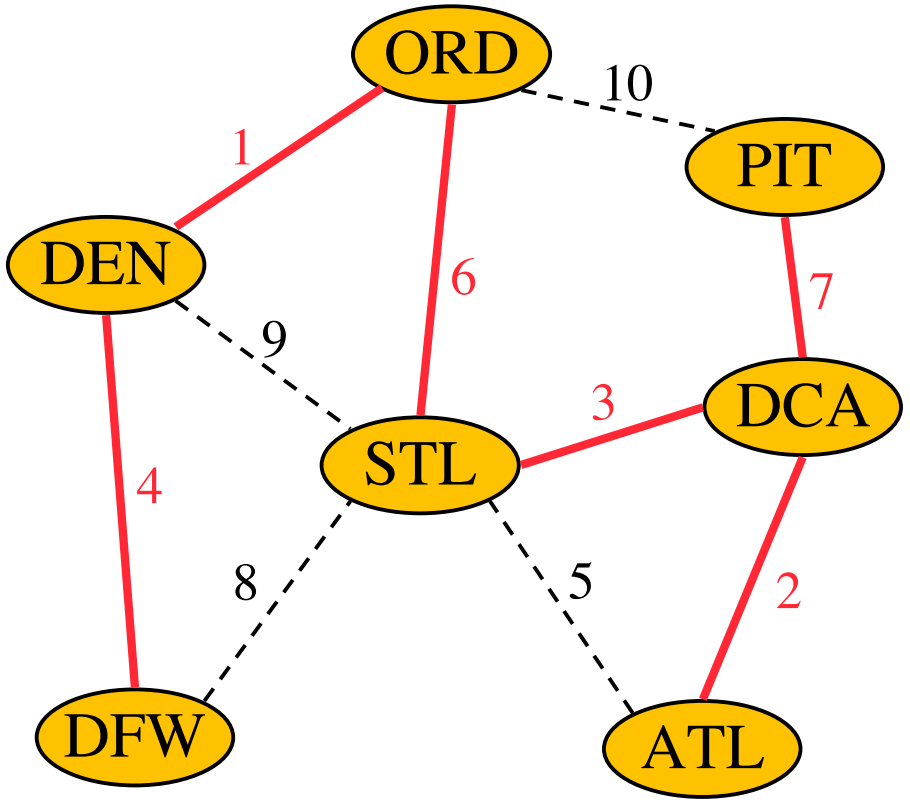
Mindste udspændende træ

Udspændende træ i en vægtet graf med mindst mulig samlet vægt

Anvendelser:

Kommunikationsnetværk

Transportnetværk



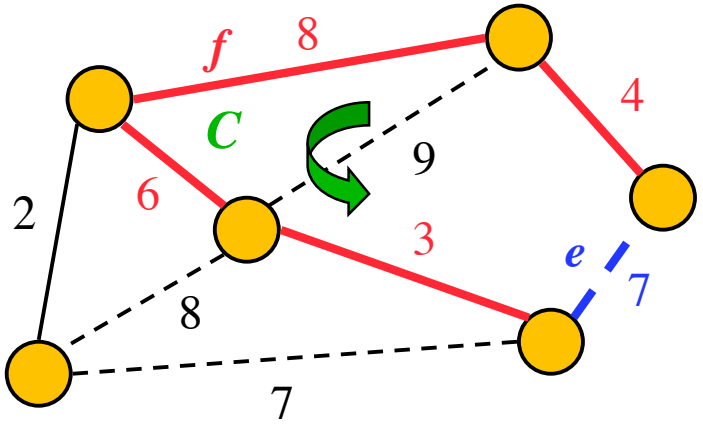
Cykelegenskab

Cykelegenskab:

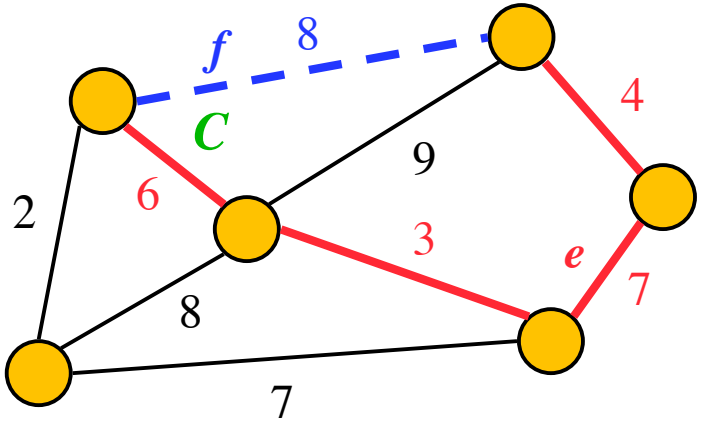
- Lad ***T*** være et mindste udspændende træ for en vægtet graf ***G***
- Lad ***e*** være en kant i **G**, som ikke er i ***T***, og lad ***C*** være den cykel, der fås ved at tilføje ***e*** til ***T***
- For enhver kant ***f*** i ***C*** gælder da, at

  weight(***f***) ≤ ***weight***(***e***)

Bevis (ved modstrid):
Hvis ***weight***(***f***) > ***weight***(***e***), kan vi opnå et udspændende med mindre vægt ved at erstatte ***e*** med ***f***



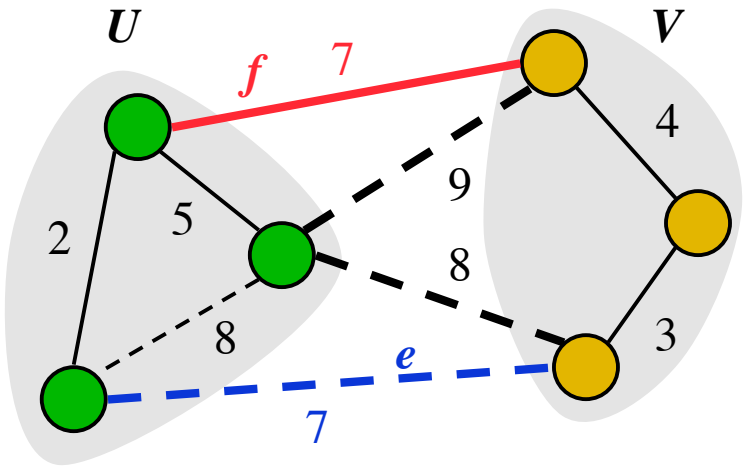
Erstatning af ***f*** med ***e*** giver et bedre udspændende træ



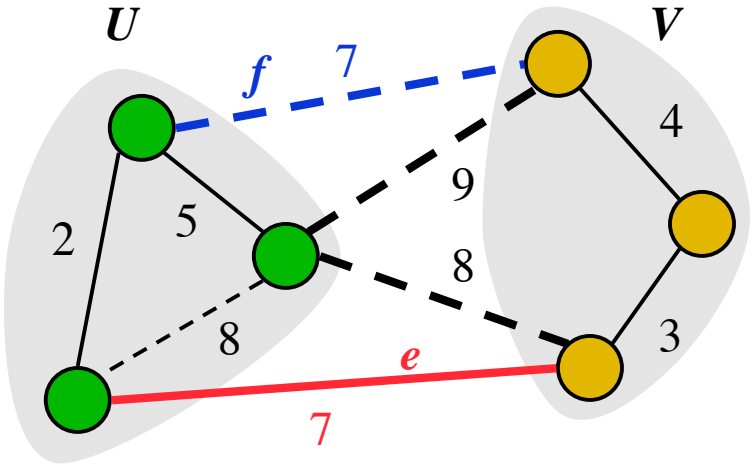
Opdelingsegenskab

Opdelingsegenskab:

- Betragt en opdeling af G's knuder i to delmængder U og V
- Lad e være en kant med mindst mulig vægt, der forbinder U og V
- Der eksister da et mindste udspændende træ, der indeholder e



Erstatning af f med e giver et andet mindste udspændende træ



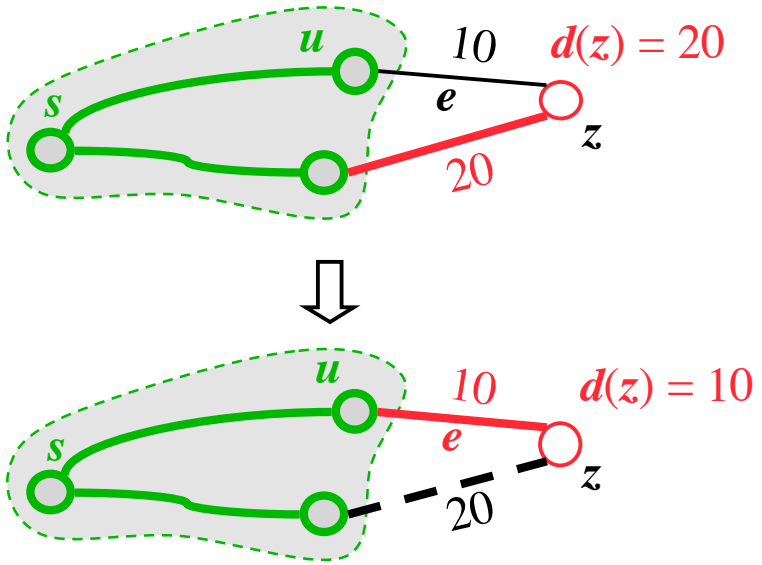
Bevis:

Lad T være et mindste udspændende træ for G. Hvis T ikke indeholder e, betragt da den cykel C, der opnås ved at tilføje e til T, og lad f være en kant, der forbinder U og V.

På grund af cykelegenskaben gælder, at $weight(f) \leq weight(e)$. Men så må $weight(f) = weight(e)$. Vi kan opnå et andet mindste udspændende træ ved at erstatte f med e

Prim-Jarnik's algoritme (1957)

- Ligner Dijkstra's algoritme (for en sammenhængende graf)
- Vi vælger en vilkårlig knude ***s*** og lader det mindste udspændende træ vokse som en sky af knuder, startende fra ***s***
- For hver knude ***v*** lagrer vi en etikette ***d***(***v***) = den mindste vægt for de kanter, der forbinder ***v*** med en knude i skyen
- I hvert trin tilføjer vi til skyen den knude ***u*** uden for skyen, der har den mindste afstandsetikette, ***d***(***u***), og opdaterer etiketterne for de knuder, der er naboer til ***u***



Prim-Jarnik's algoritme (forsat)

En prioritetskø indeholder de knuder, der er uden for skyen

Nøgle: afstand

Element: knude

- Locator-baserede metoder:

  insert(*k*,*e*)

  returnerer en locator

  replaceKey(*l*,*k*)

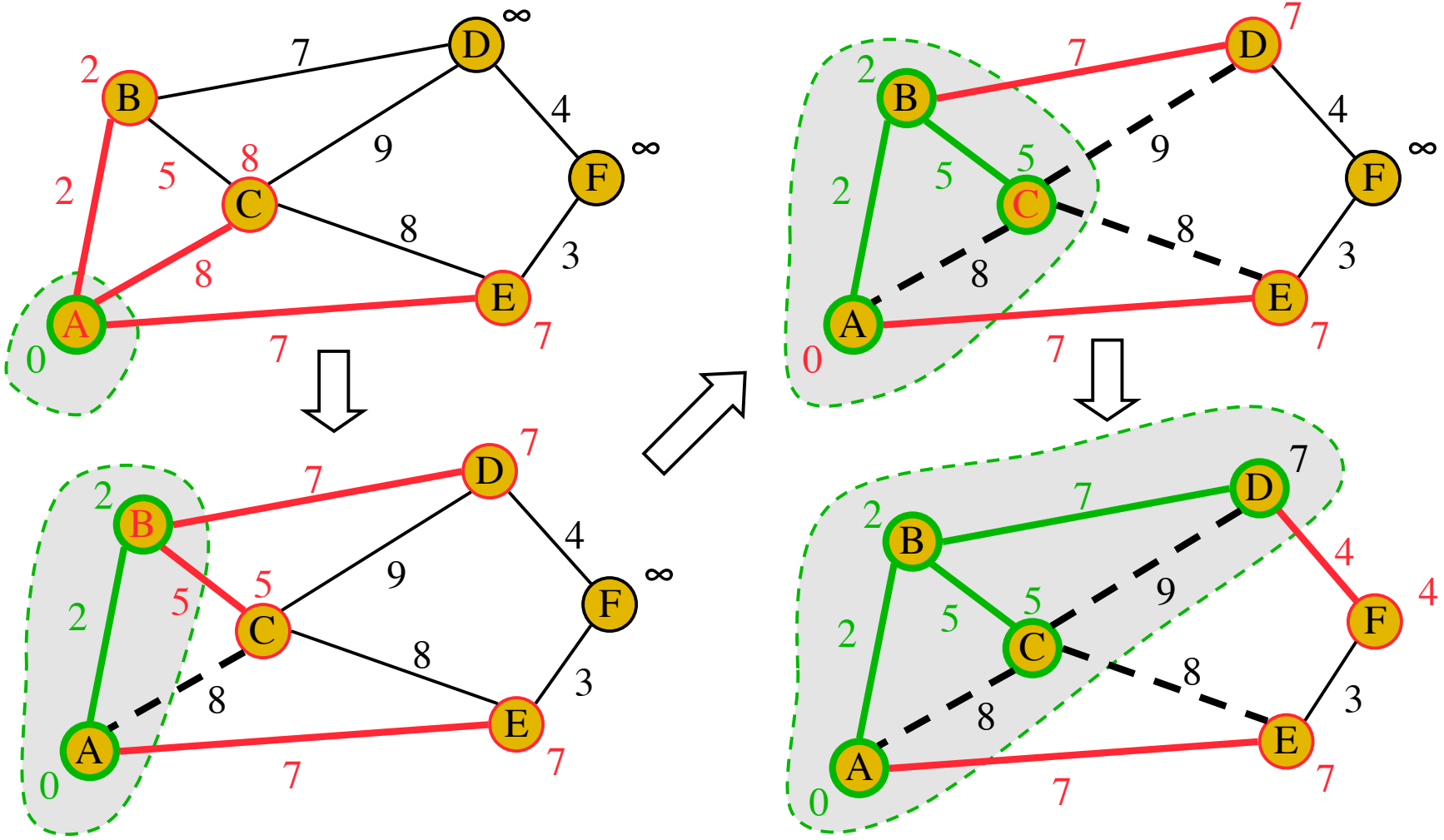
  ændrer nøglen for et emne

- For hver knude ***v*** gemmes tre etiketter:

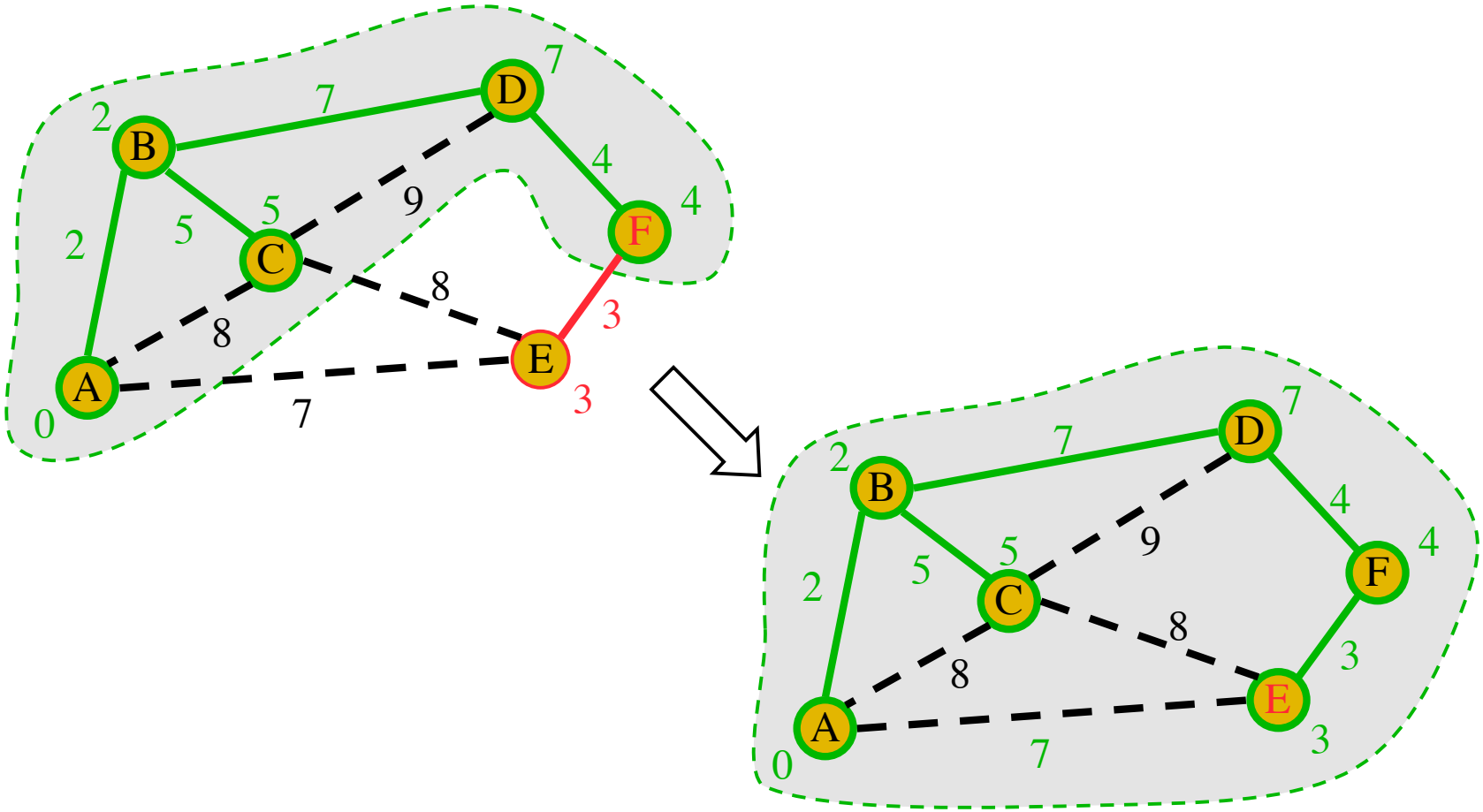
  afstand, ***d***(***v***)
  locator i prioritetskøen
  forældrekanten

```
Algorithm PrimJarnikMST(G)
    Q ← new heap-based priority queue
    s ← a vertex of G
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
        setParent(v, ∅)
        l ← Q.insert(getDistance(v), v)
        setLocator(v,l)
    while ¬Q.isEmpty()
        u ← Q.removeMin()
        for all e ∈ G.incidentEdges(u)
            z ← G.opposite(u,e)
            r ← weight(e)
            if r < getDistance(z)
                setDistance(z,r)
                setParent(z,e)
                Q.replaceKey(getLocator(z),r)
```

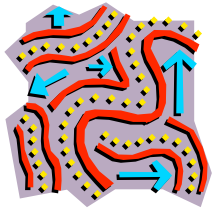
Eksempel



Eksempel (fortsat)



Analyse



- Grafoperationer:

  Metoden **incidentEdges** kaldes en gang for hver knude

- Etiketteoperationer:

  Vi sætter eller henter afstands- og locator-etiketterne for en knude z
  $O(\deg(z))$ gange. Hver sådan operation tager O(1) tid

- Prioritetskøoperationer:

  Hver knude indsættes en gang i og fjernes en gang fra prioritetskøen

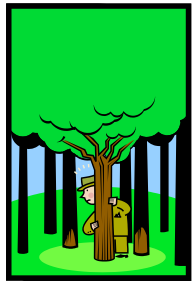
  Hver indsættelse og fjernelse tager $O(\log n)$ tid

  Nøglen for en knude w i prioritetetskøen ændres højst $\deg(w)$ gange,
  hvor hver ændring tager $O(\log n)$ tid

- Prim-Jarnik's algoritme kører i $O((n + m) \log n)$ tid, hvis grafen er repræsenteret ved en nabolistestruktur (husk, at $\sum_v \deg(v) = 2m$)

- Køretiden kan også udtrykkes som $O(m \log n)$, da grafen er sammenhængende

Kruskal's algoritme (1956)



- En prioritetskø indeholder **kanterne**
  - Nøgle: vægt
  - Element: kant
- Ved afslutningen af algoritmen står vi tilbage med
  - (1) en sky, der omfatter et mindste udspændende træ
  - (2) et træ, ***T***, der er det søgte mindste udspændende træ

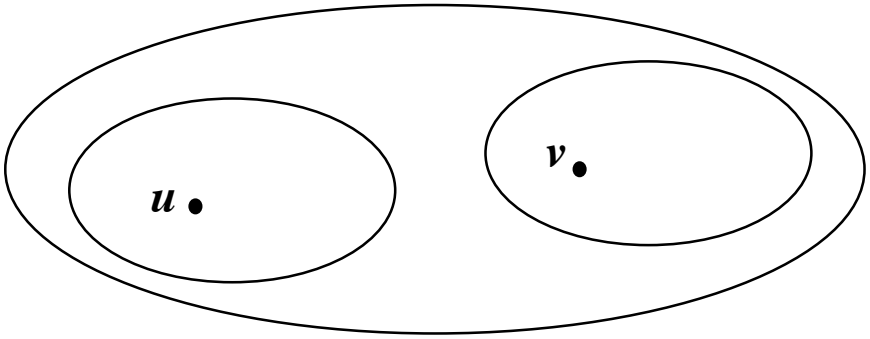
Algorithm *KruskalMST*(***G***)
  for each vertex ***V*** in ***G*** **do**
    define a ***Cloud(v)*** of {***v***}
  let ***Q*** be a priority queue.
  Insert all edges into ***Q*** using their weights as the key
  T ← ∅
  while ***T*** has fewer than ***n***-1 edges **do**
    edge ***e*** = ***Q.removeMin()***
    Let ***u***, ***v*** be the endpoints of ***e***
    if ***Cloud(v)*** ≠ ***Cloud(u)*** **then**
      Add edge ***e*** to ***T***
      Merge ***Cloud(v)*** and ***Cloud(u)***
  return ***T***

Datastruktur til Kruskal's algoritme

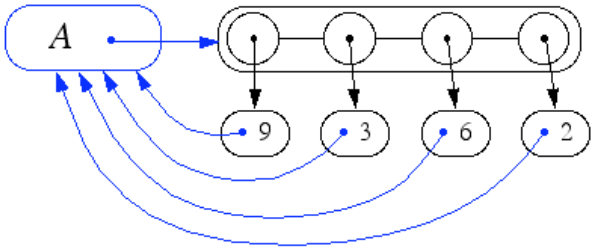
Algoritmen vedligeholder en skov af træer. En kant accepteres, hvis den forbinder to forskellige træer

Vi har brug for en datastruktur, der vedligeholder en opdeling, d.v.s. en samling af disjunkte mængder med følgende to operationer:

find(***u***): returner den mængde, der indeholder ***u***
union(***u***,***v***): erstat mængderne, som ***u*** og ***v*** tilhører med deres foreningsmængde



Repræsentation af en opdeling



Hver mængde repræsenteres ved en sekvens

Hvert element har en reference til den mængde, elementet tilhører

- Operationen find(***u***) tager ***O***(1) tid og returnerer den mængde, som ***u*** er medlem af
- I operationen union(***u***,***v***), flytter vi elementer fra den mindste mængde til sekvensen for den største mængde og opdaterer deres referencer
- Tiden for operationen union(***u***,***v***), er min(n_u,n_v), hvor n_u og n_v er størrelserne af mængderne, der indeholder ***u*** og ***v***

Hver gang et element flyttes, placeres det i en mængde af mindst den dobbelte størrelse
Derfor vil et element højst blive flyttet log ***n*** gange

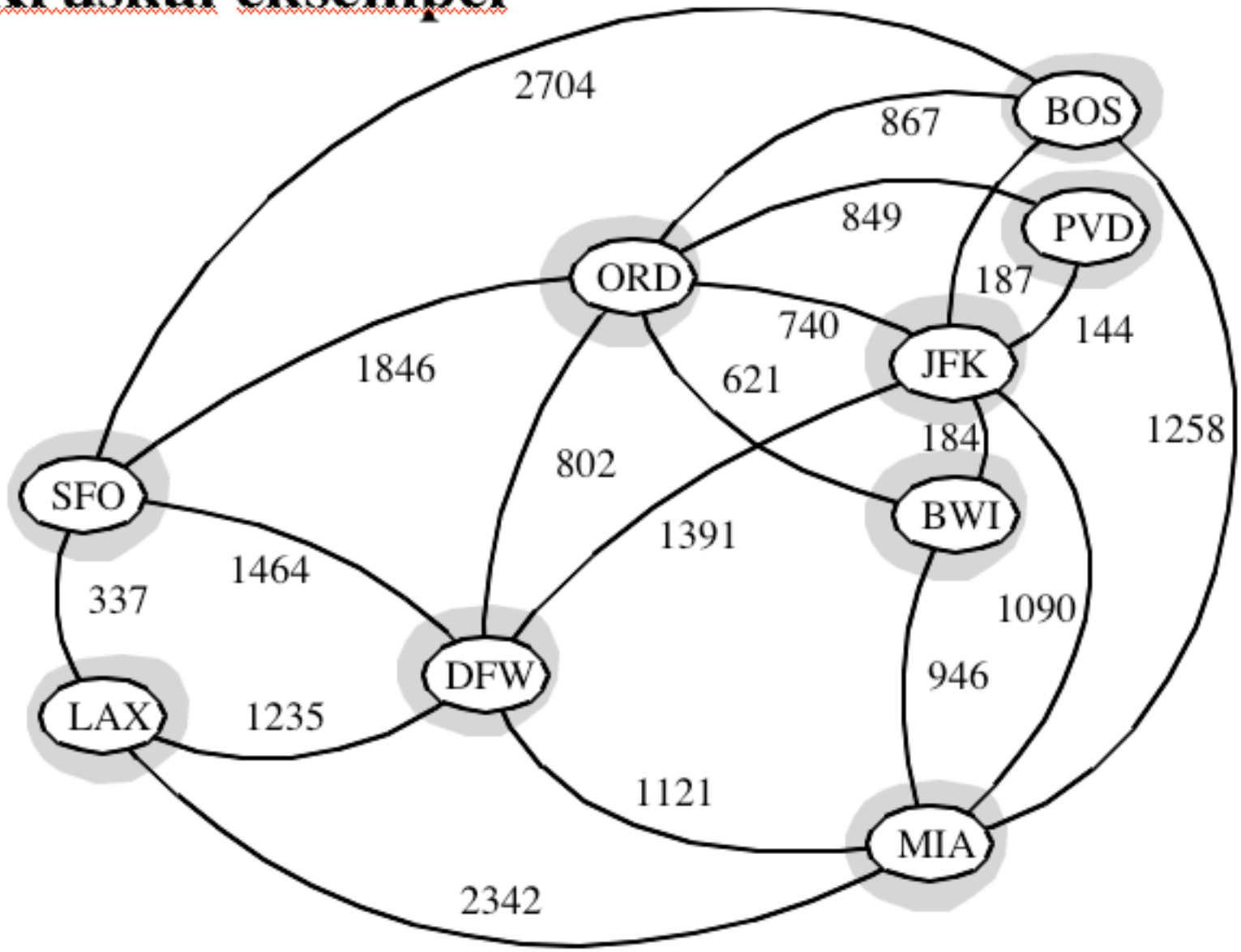
Opdelingsbaseret implementation

En opdelingsbaseret implementation af Kruskal's algoritme udfører sammenføjning af skyer ved kald af union og afgørelse af mængdetilhørsforhold ved kald af find

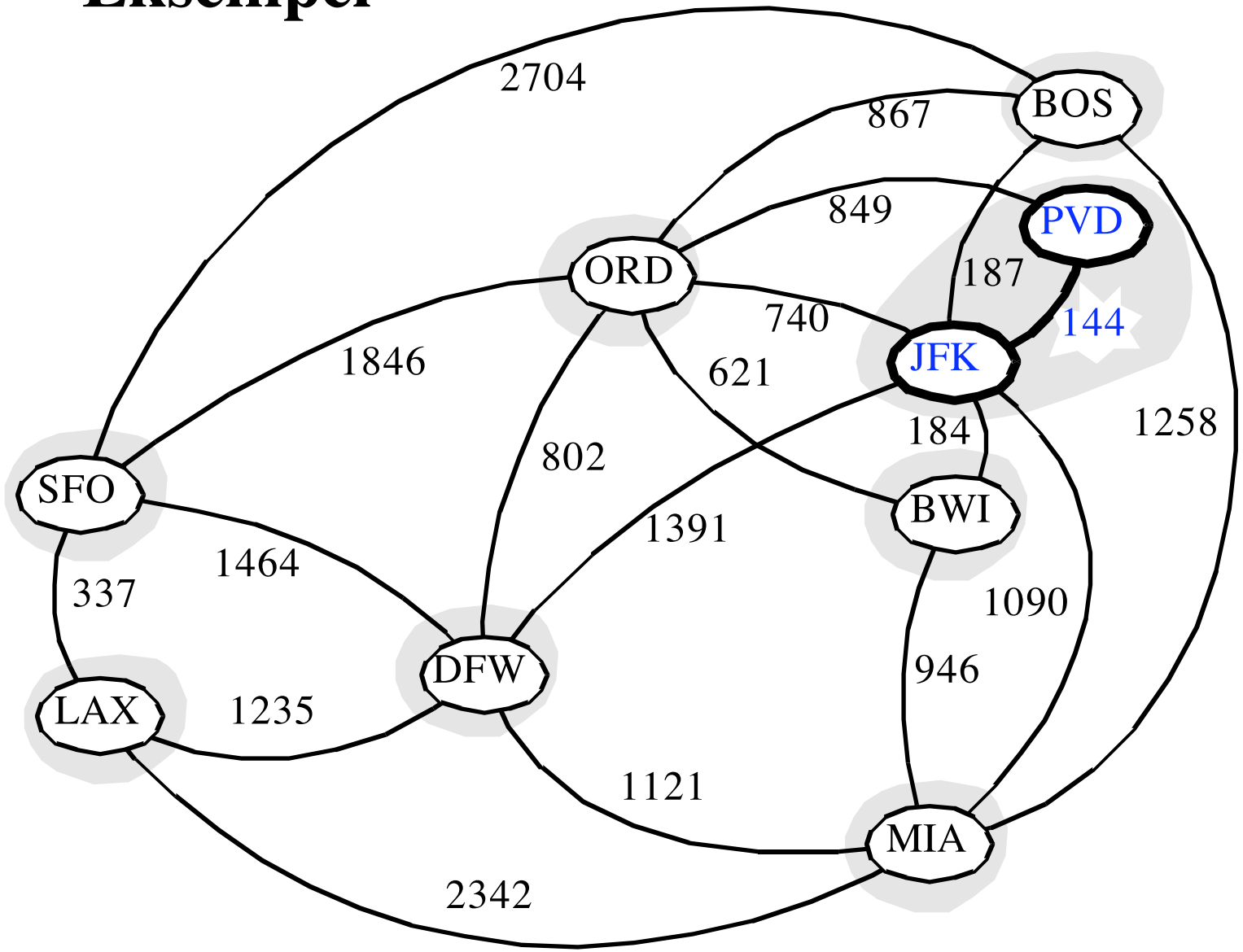
Algorithm Kruskal(***G***):
  Input: A weighted graph ***G***.
  Output: An MST ***T*** for ***G***.
Let ***P*** be a partition of the vertices of ***G***, where each vertex forms a separate set
Let ***Q*** be a priority queue storing the edges of ***G***, sorted by their weights
Let ***T*** be an initially-empty tree
while ***Q*** is not empty **do**
  (***u***,***v***) ← ***Q***.removeMinElement()
  if ***P***.find(***u***) != ***P***.find(***v***) **then**
    Add (***u***,***v***) to ***T***
    P.union(***u***,***v***)
return ***T***

Køretid: $O((n+m)\log n)$

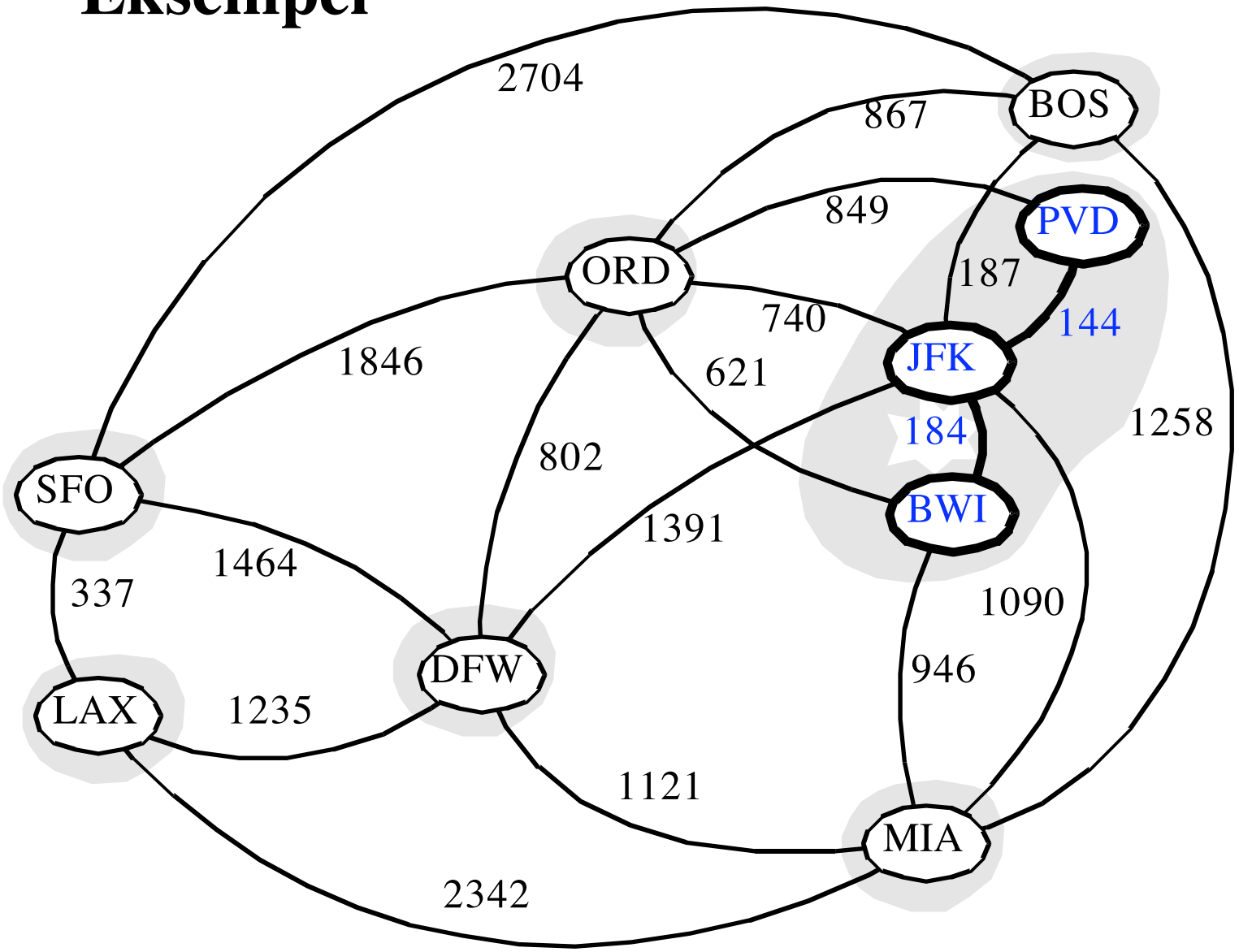
Kruskal eksempel



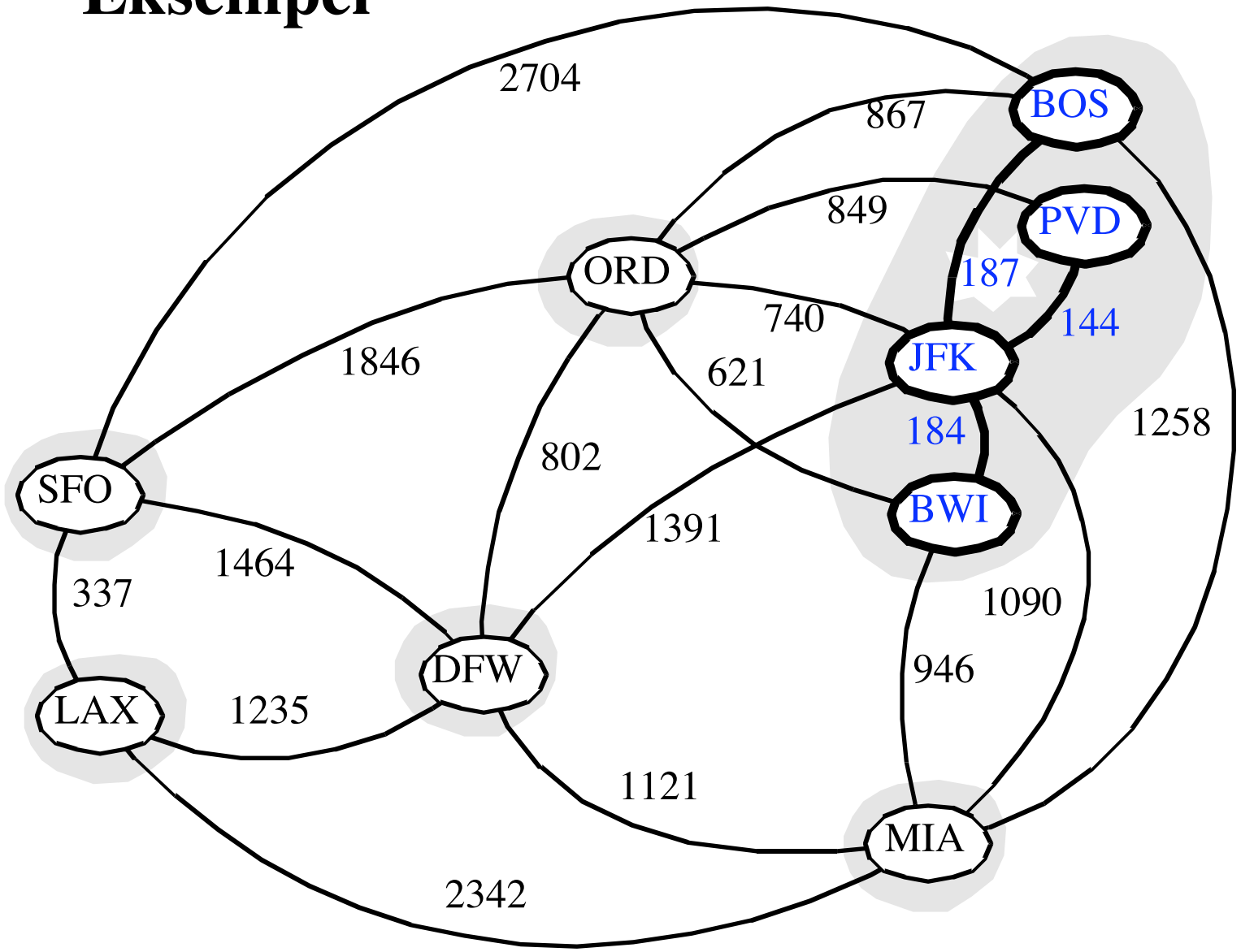
Eksempel


2704
867
BOS
849
PVD
ORD
187
740
144
JFK
1846
621
1258
184
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

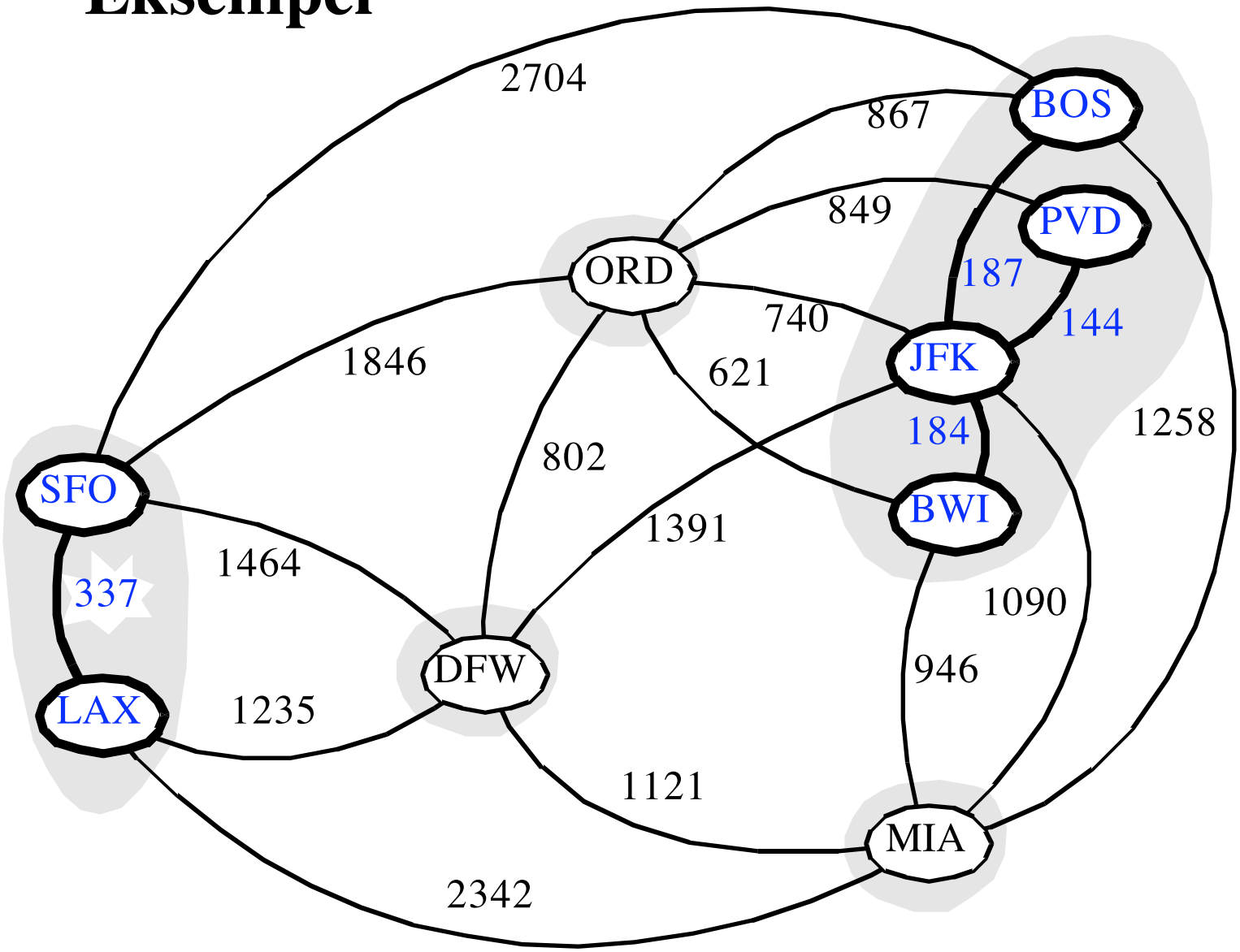
Eksempel


2704
867
BOS
849
PVD
ORD
187
740
144
1846
621
JFK
184
1258
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

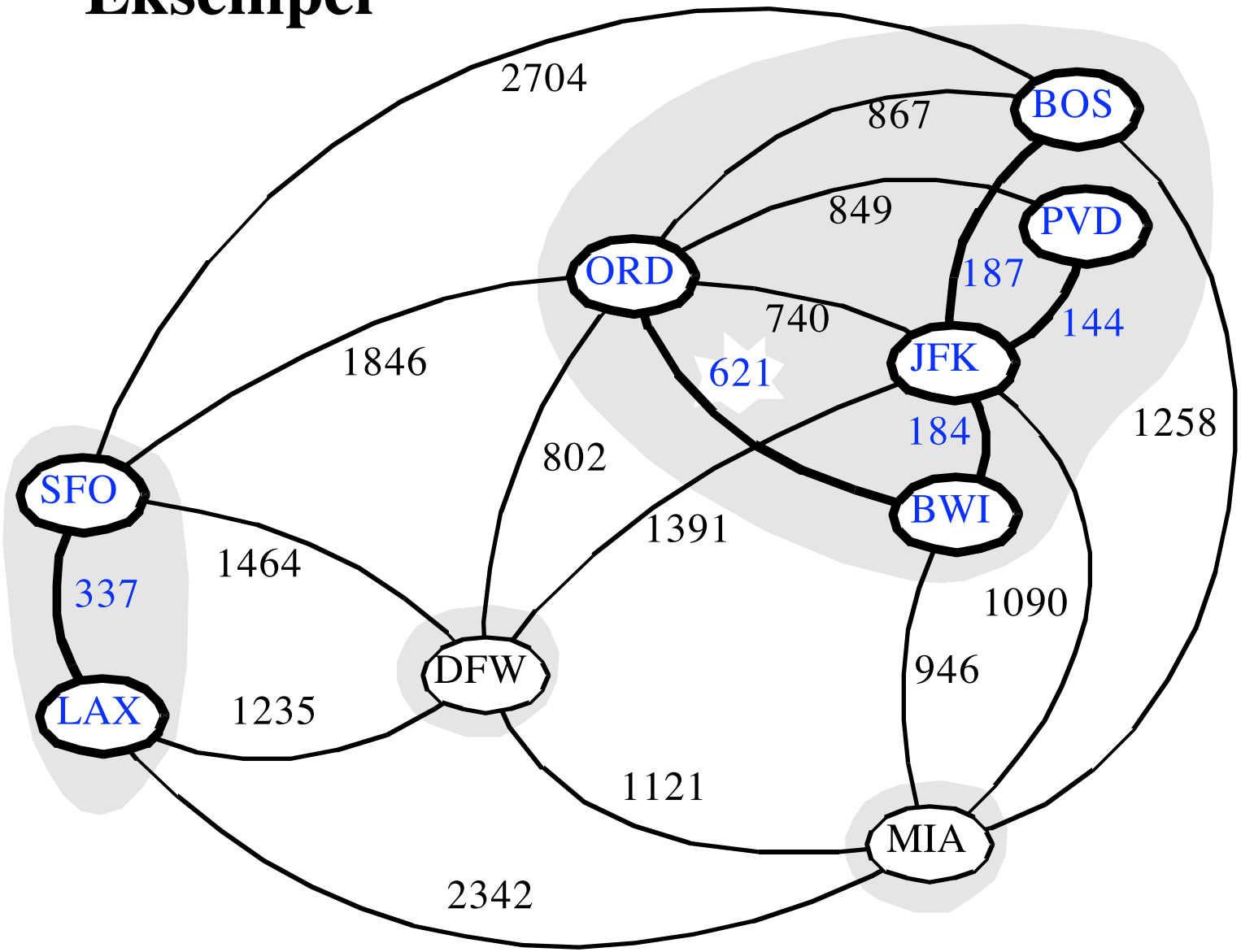
Eksempel


2704
867
849
187
740
144
1846
621
1258
184
802
1391
1464
337
1090
946
1235
1121
2342
BOS
PVD
ORD
JFK
SFO
BWI
DFW
LAX
MIA

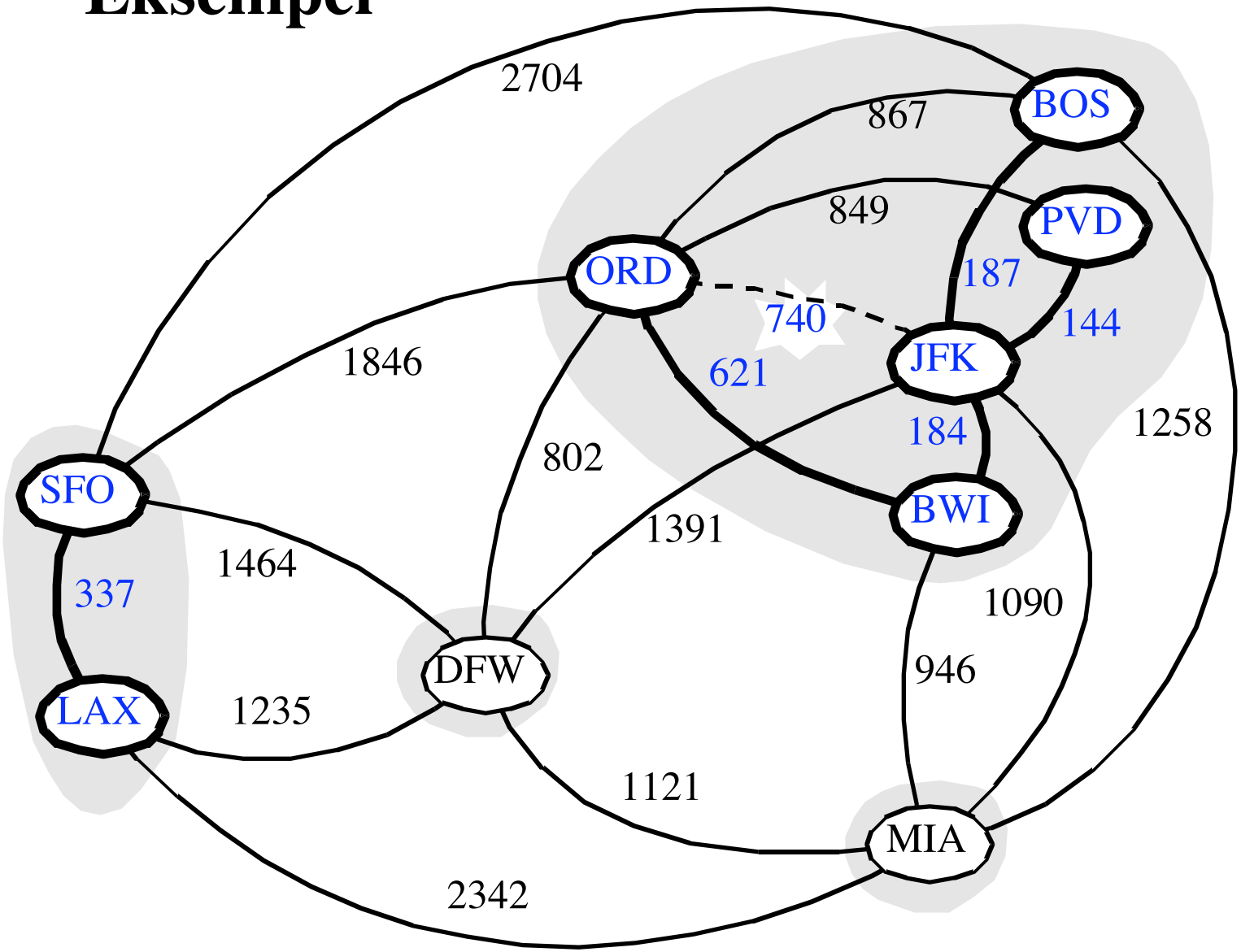
Eksempel


2704
867
BOS
849
PVD
ORD
187
740
144
1846
621
JFK
1258
184
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

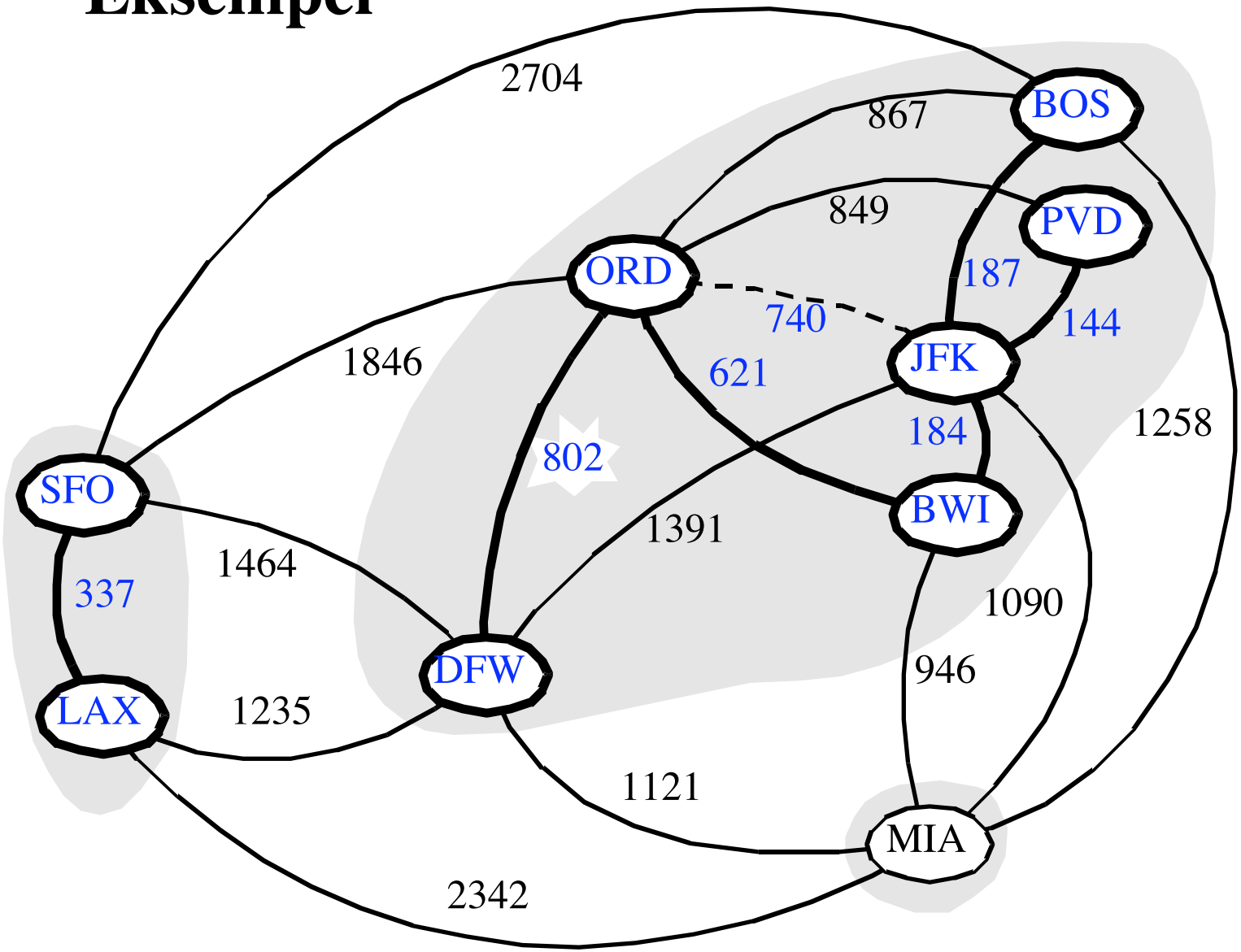
Eksempel


2704
867
BOS
849
PVD
ORD
187
740
144
1846
621
JFK
1258
184
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

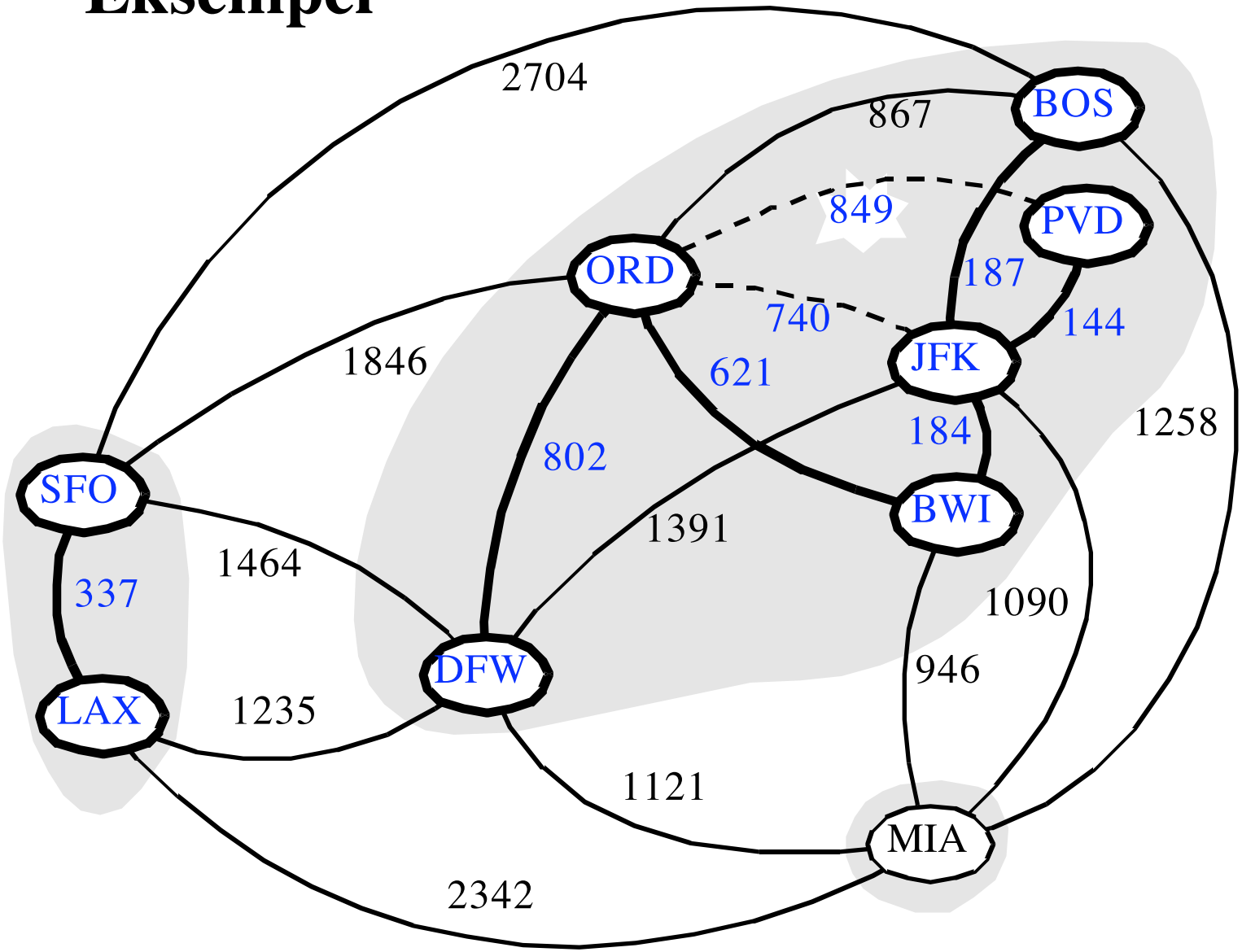
Eksempel


2704
867
BOS
849
PVD
ORD
187
740
144
1846
621
JFK
184
1258
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

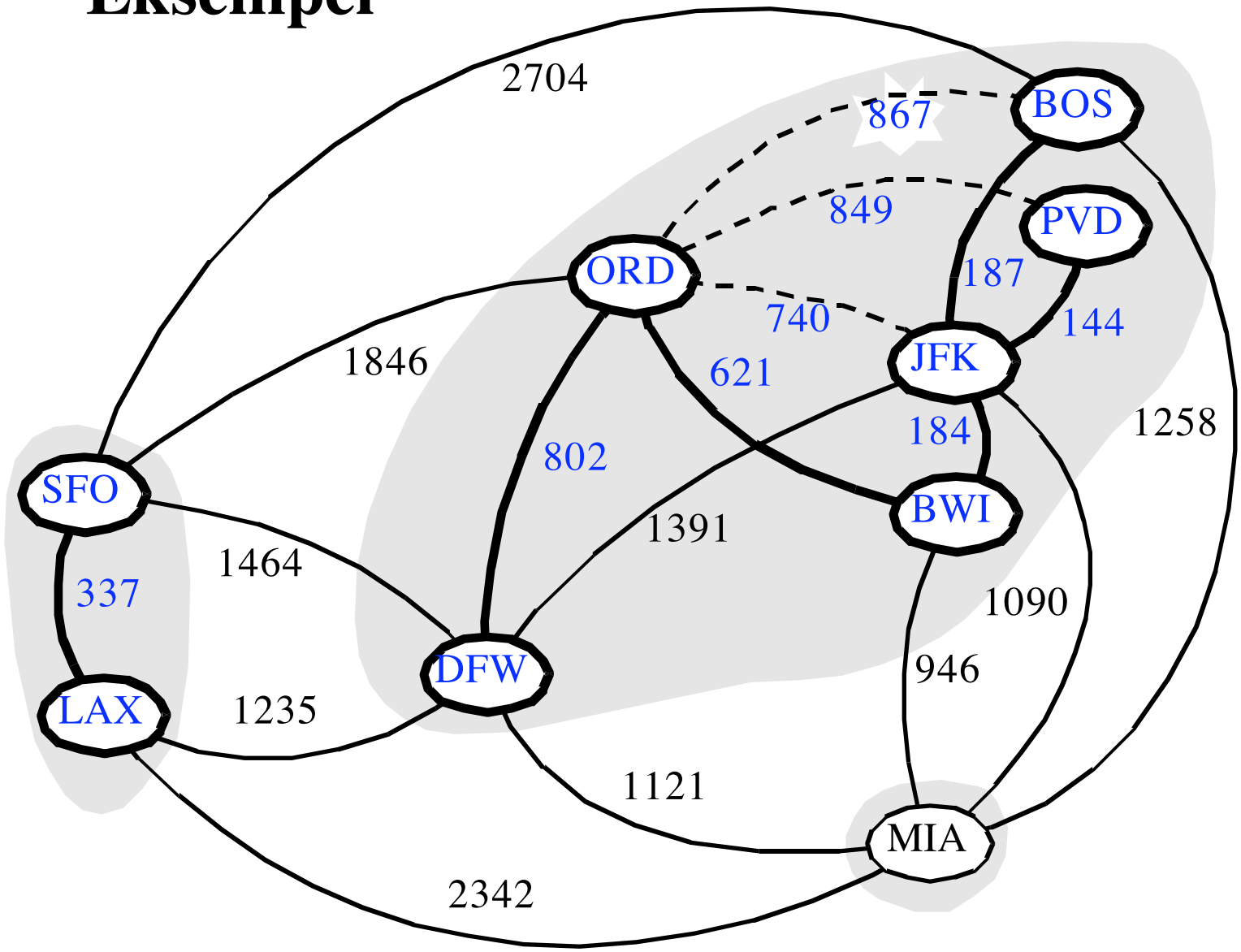
Eksempel


2704
867
BOS
849
PVD
ORD
187
740
144
1846
621
JFK
184
1258
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

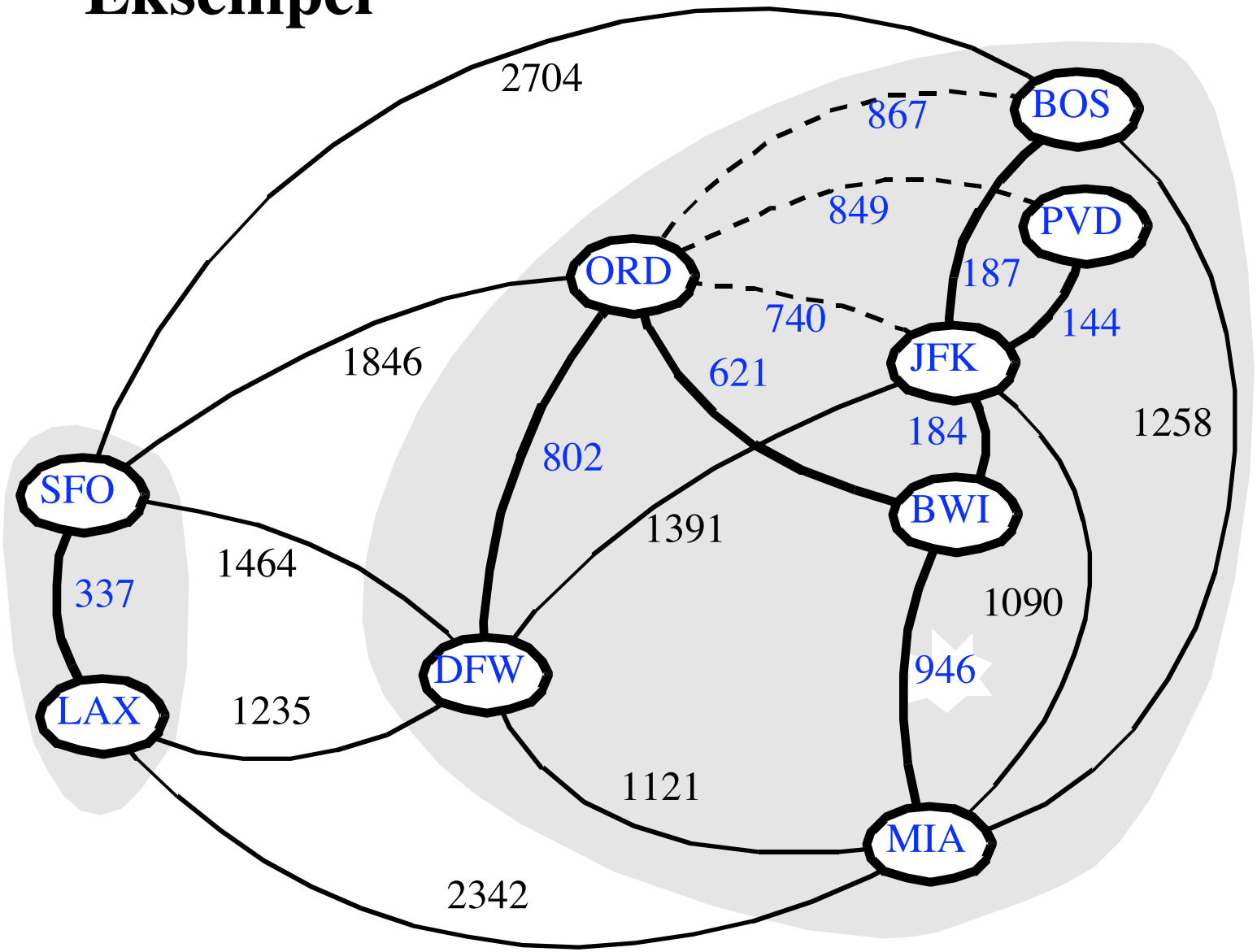
Eksempel


2704
867
BOS
849
PVD
ORD
187
740
144
1846
621
JFK
1258
184
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

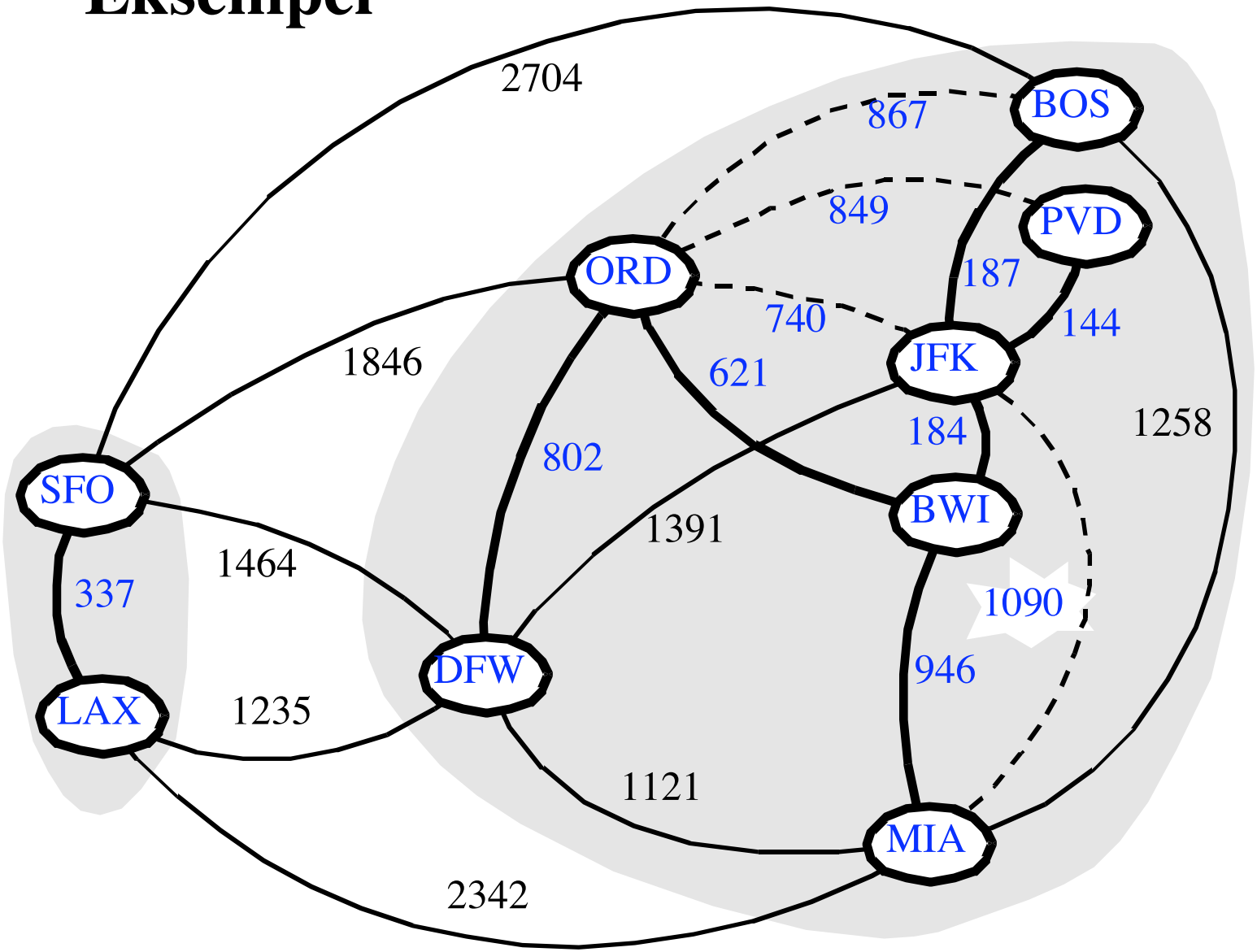
Eksempel


2704
867
BOS
849
PVD
ORD
187
740
144
JFK
1846
621
184
1258
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

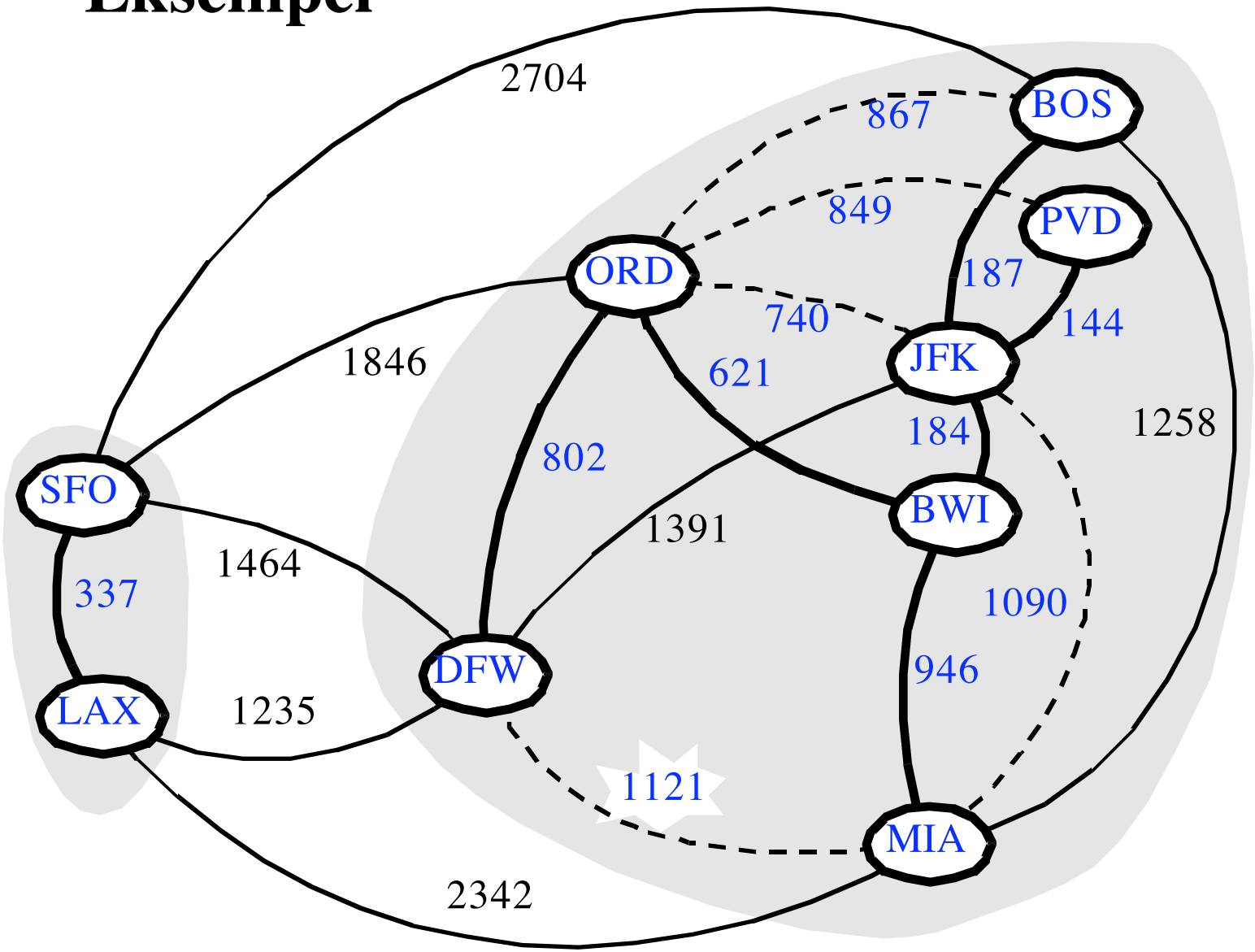
Eksempel


2704
867
BOS
849
PVD
ORD
187
740
144
1846
621
JFK
1258
184
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

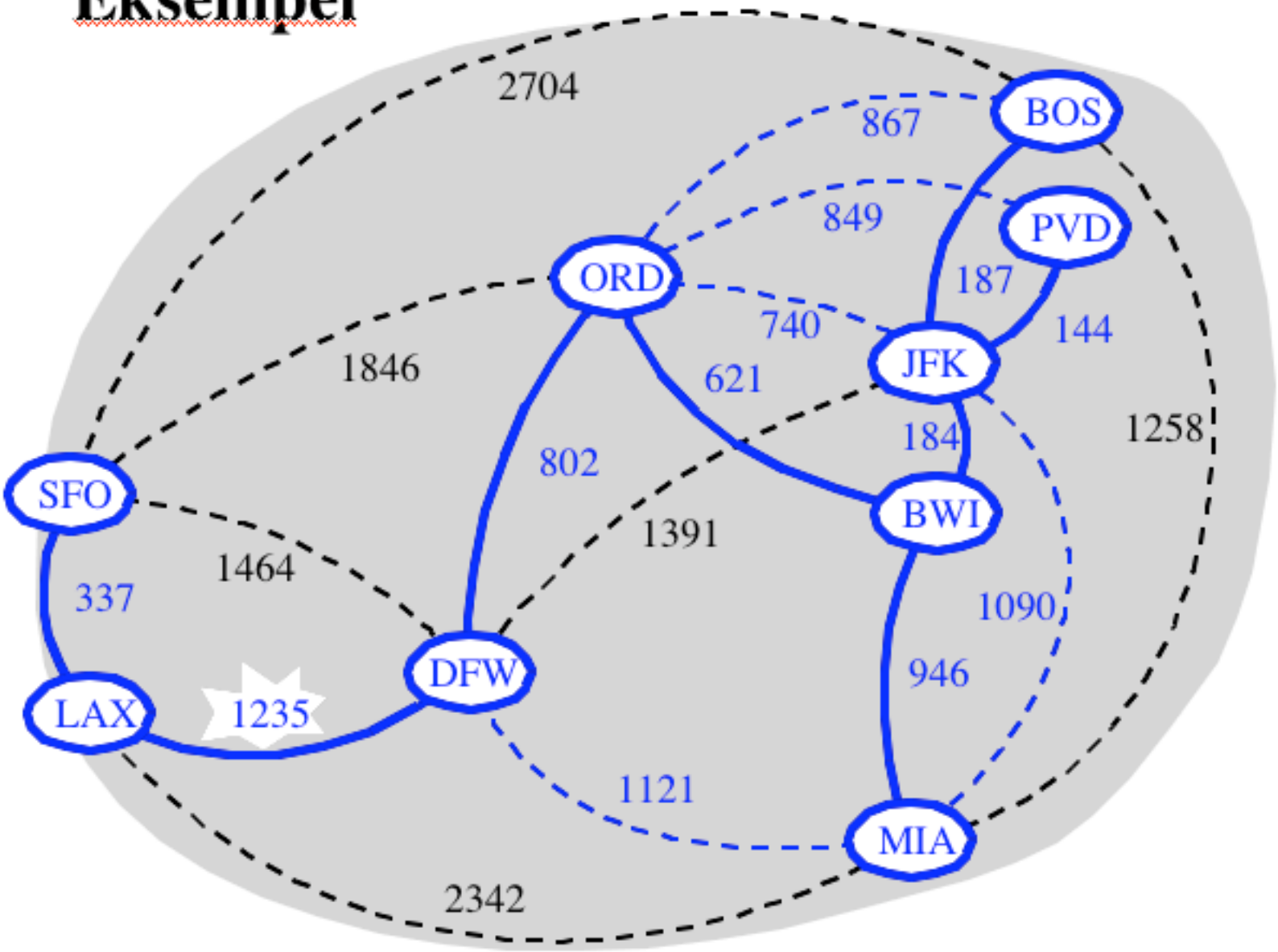
Eksempel


2704
867
BOS
849
PVD
ORD
187
740
144
1846
621
JFK
1258
184
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

Eksempel


2704
867
BOS
849
PVD
ORD
187
740
144
1846
621
JFK
1258
184
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

Eksempel


2704
867
849
187
740
144
1846
621
184
1258
802
1391
1464
337
1090
946
1235
1121
2342
BOS
PVD
ORD
JFK
SFO
BWI
DFW
LAX
MIA

Baruvka's algoritme (1926)

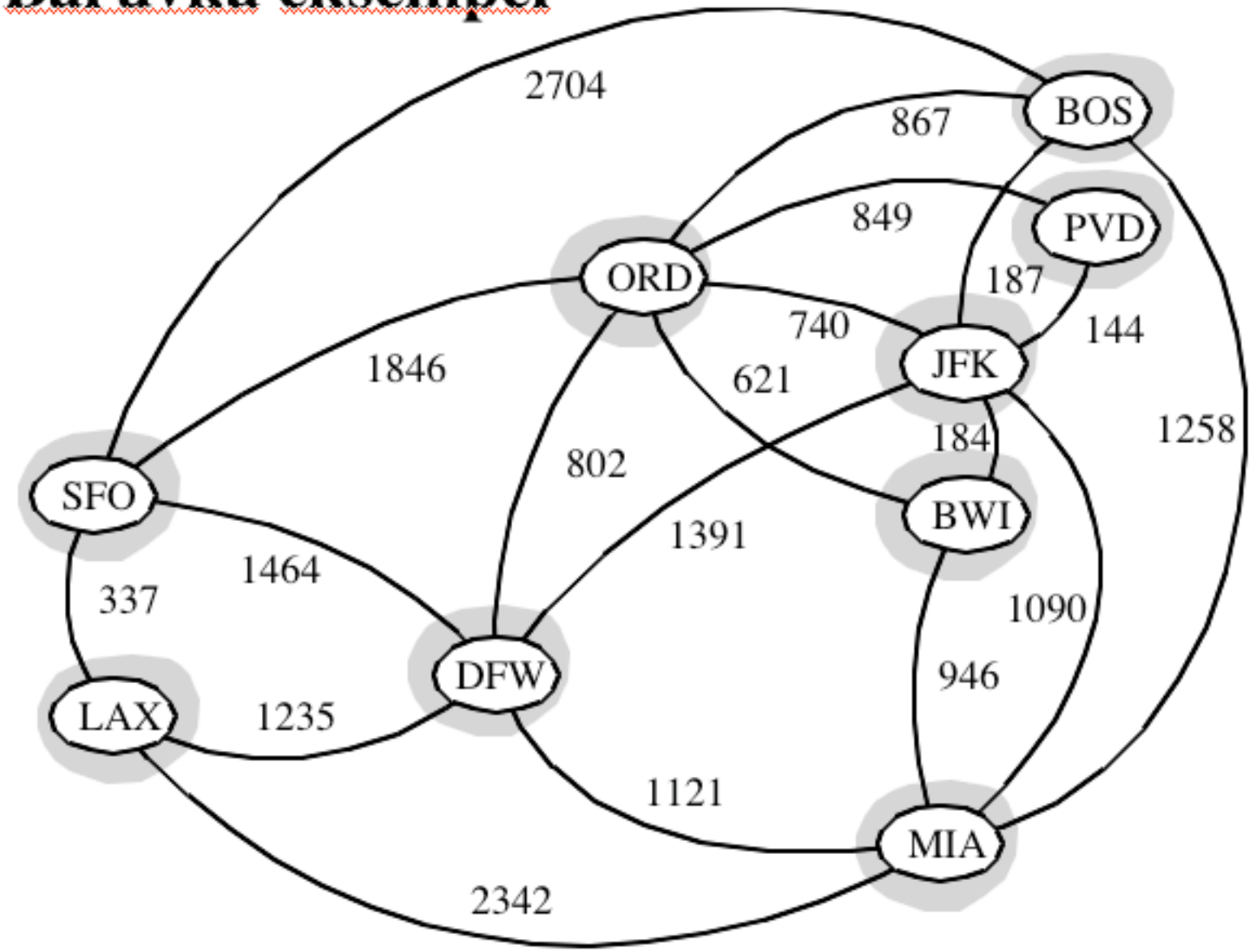
- Ligner Kruskal's algoritme. Baruvka's algoritme udvider mange "skyer" på en gang

```
Algorithm BaruvkaMST(G)
   T ← V  {just the vertices of G}
   while T has fewer than n-1 edges do
      for each connected component C in T do
         Let edge e be the smallest-weight edge from C to another component in T
         if e is not already in T then
            Add edge e to T
   return T
```

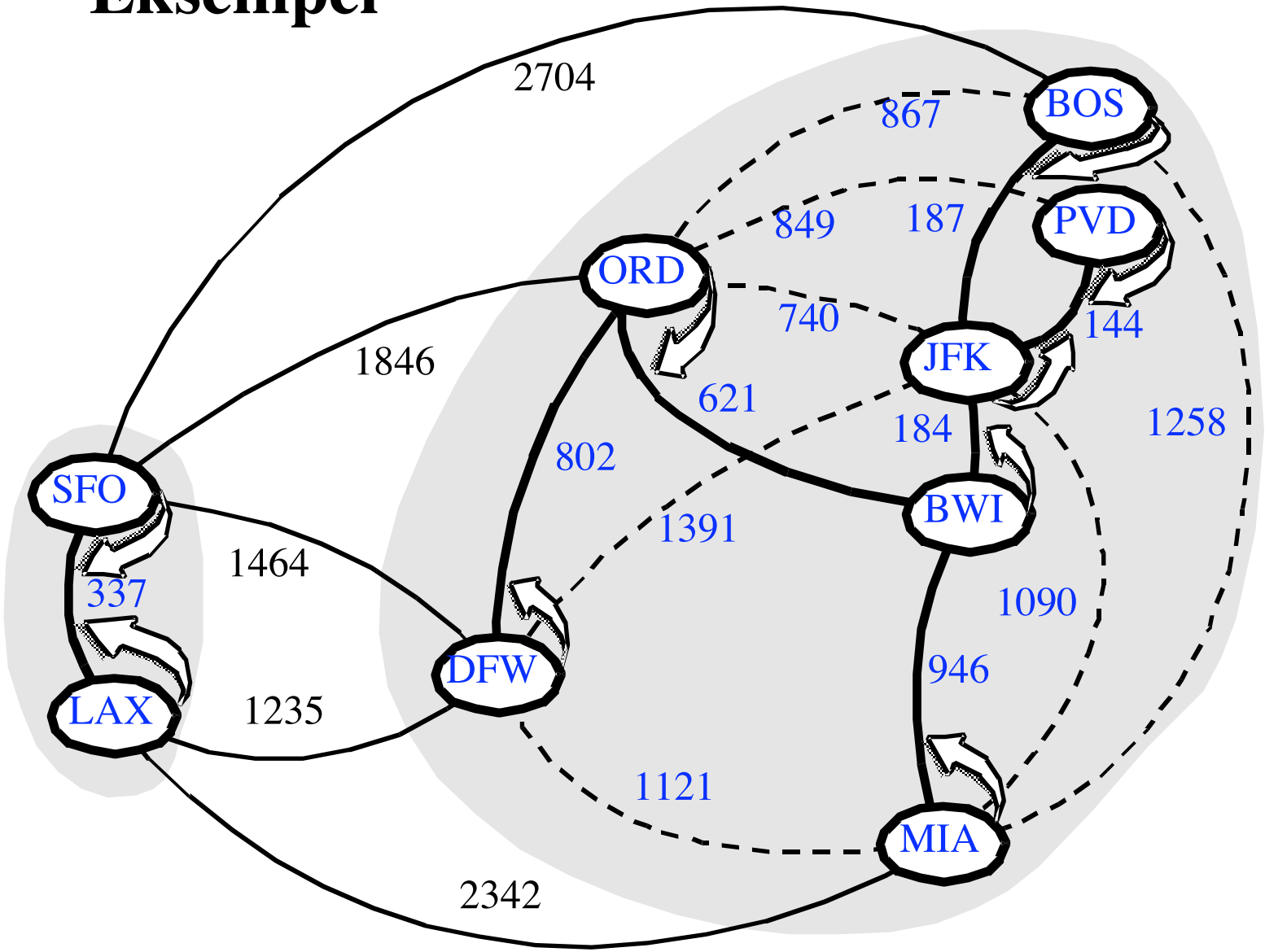
- Hver iteration af while-løkken halverer antallet af sammenhængende komponenter

- Køretiden er $O(m \log n)$

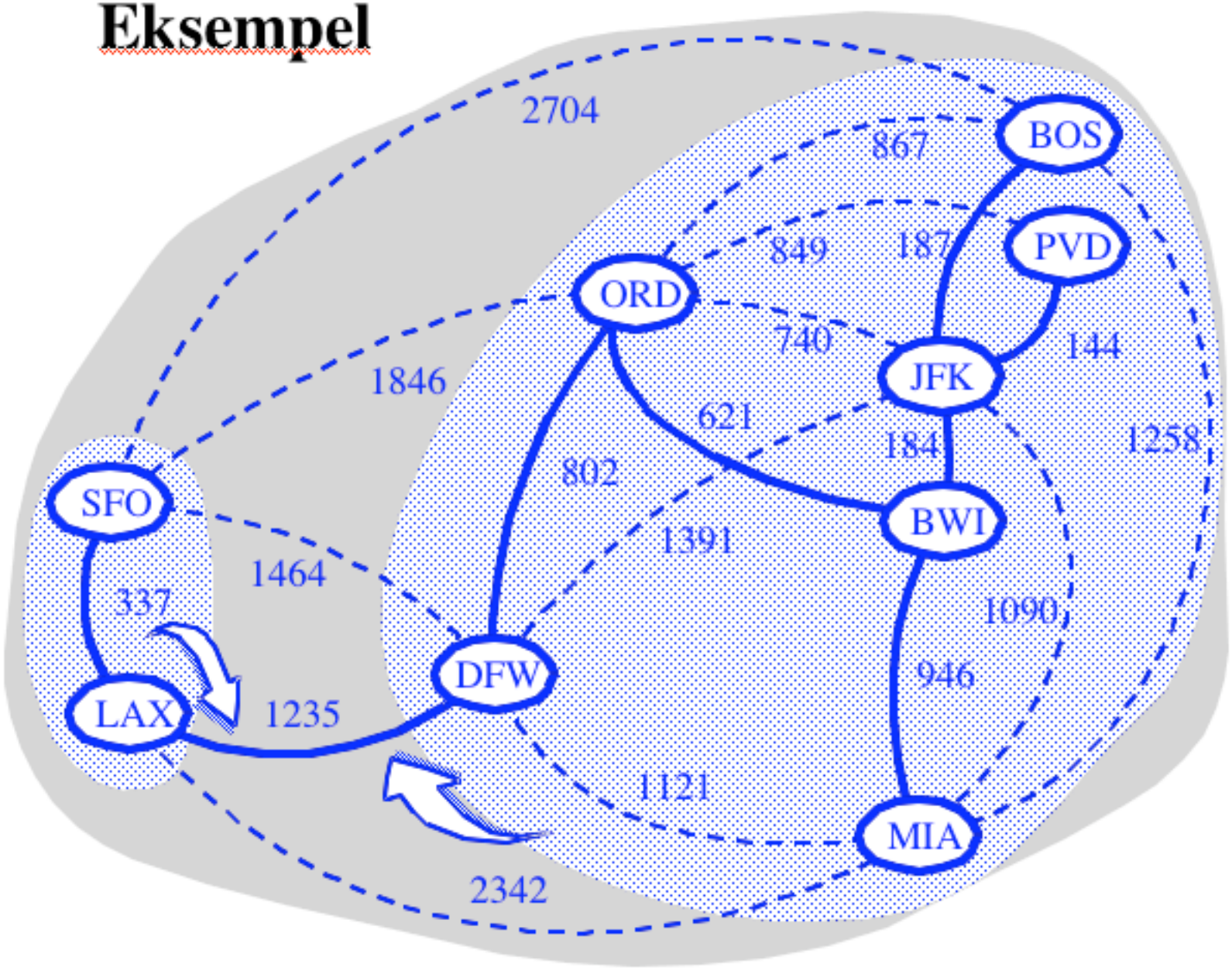
Baruvka eksempel



Eksempel


2704
867
BOS
849
187
PVD
ORD
740
144
1846
JFK
621
184
1258
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

Eksempel


2704
867
BOS
849
187
PVD
ORD
740
144
1846
JFK
621
184
1258
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342