Greedy Algorithms

Often when trying to find the optimal solution to some problem you need to consider all your possible choices and how they might interact with other choices down the line.

But sometimes you don't. Sometimes you can just take what looks like the best option for now and repeat.

Greedy Algorithms

General Algorithmic Technique:

1. Find decision criterion
2. Make best choice according to criterion
3. Repeat until done

Surprisingly, this sometimes works.

Greed choice property:

a sequence of locally optimal (greedy) choices

$\Rightarrow$ an optimal solution

Interval Scheduling

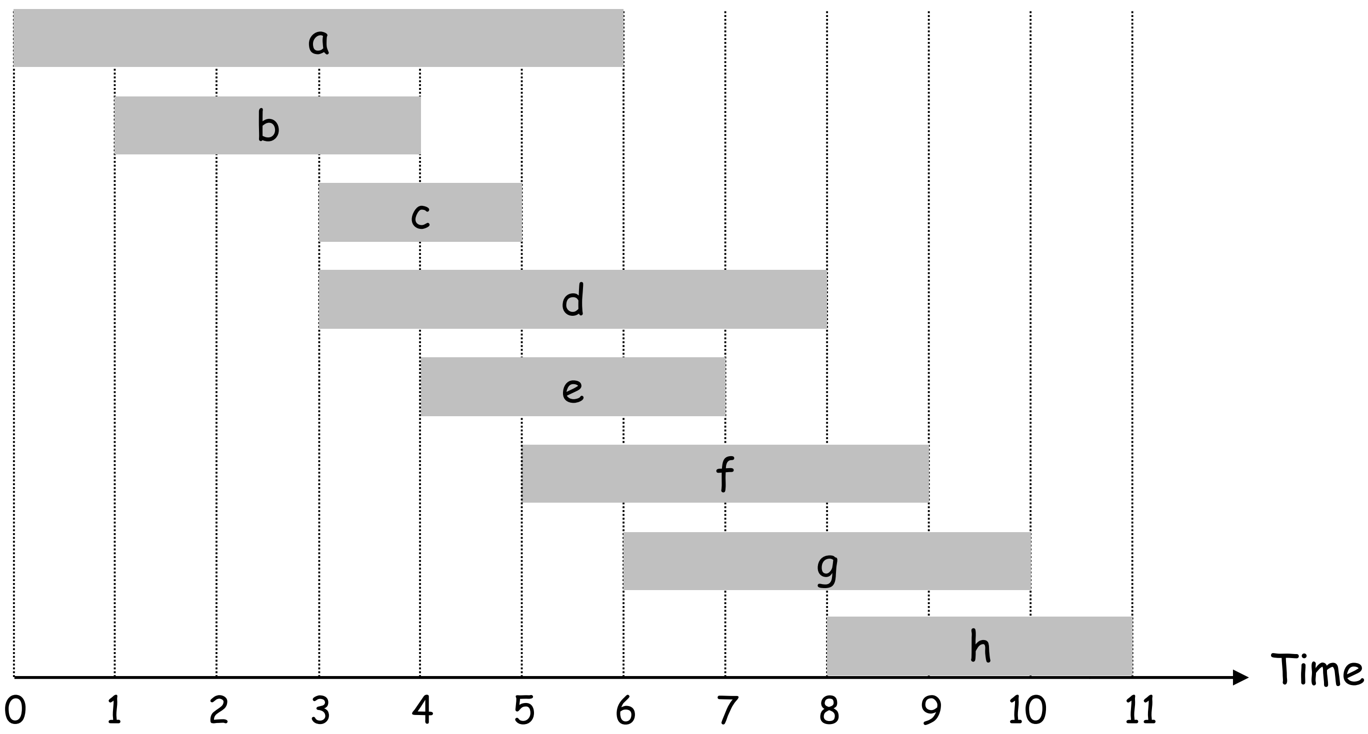
Problem: Given a collection C of intervals, find a subset S ⊆ C so that:

1. No two intervals in S overlap.
2. Subject to (1), |S| is as large as possible.

Interval Scheduling

Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

- [Earliest start time]
  Consider jobs in ascending order of start time s_j.

- [Shortest interval]
  Consider jobs in ascending order of interval length $f_j - s_j$.

- [Fewest conflicts]
  For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j.

- [Earliest finish time]
  Consider jobs in ascending order of finish time f_j.

Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

breaks earliest start time

breaks shortest interval

breaks fewest conflicts

Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

jobs selected

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Implementation. $O(n \log n)$.

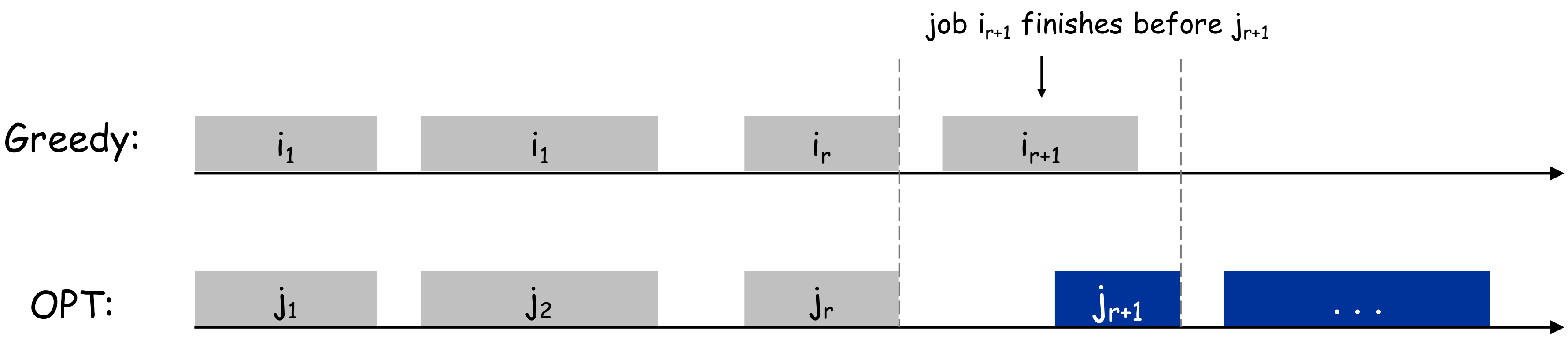
- Remember job j^* that was added last to A.
- Job j is compatible with A if $s_j \geq f_{j^*}$.

Interval Scheduling: Analysis

Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy algorithm.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r. That is, let r+1 be the first meeting where $i_{r+1} \neq j_{r+1}$.



By the design of the algorithm, we have that i_{r+1} end before j_{r+1}, and therefore i_{r+1} ends before j_{r+2} starts. Hence, the new solution

$OPT' = \{i_1, i_2, \ldots, i_r, i_{r+1}, j_{r+2}, \ldots, j_r\}$ is also a valid solution

Remarks

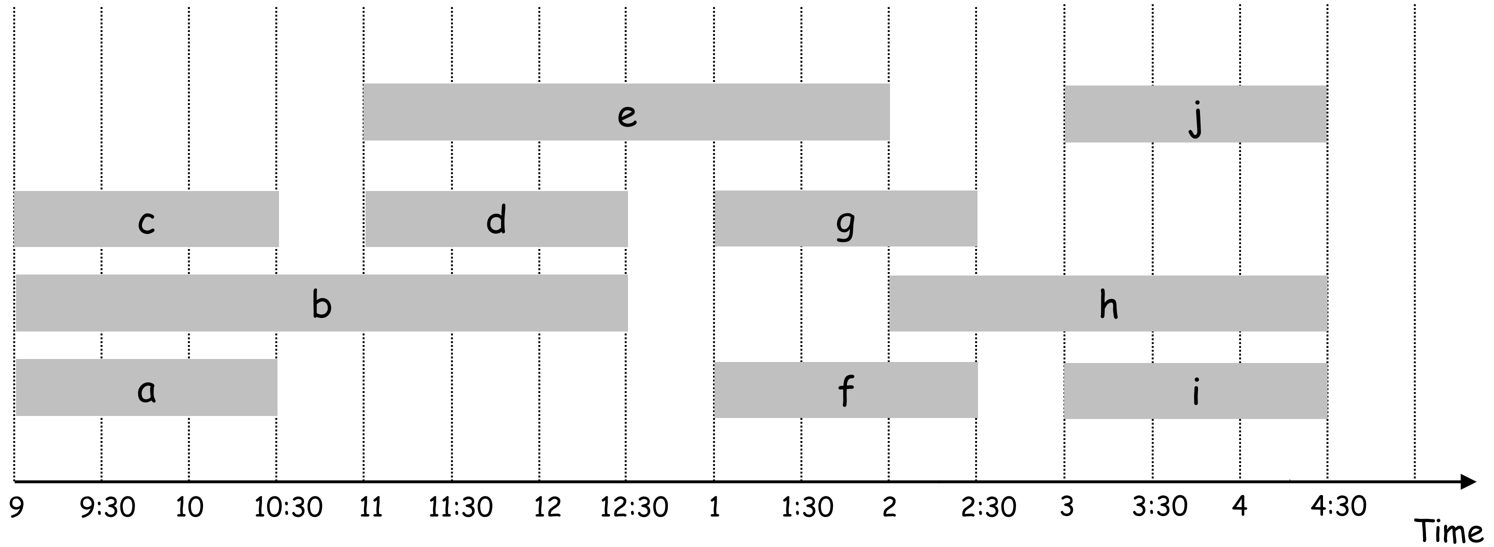
- In the Interval Scheduling Problem, there is a single resource and many requests in the form of time intervals, so we must choose which requests to accept and which to reject.
- A related problem arises if we have many identical resources available and we wish to schedule all the requests using as few resources as possible. Because the goal here is to partition all intervals across multiple resources, we will refer to this as the Interval Partitioning Problem

Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses 4 classrooms to schedule 10 lectures.

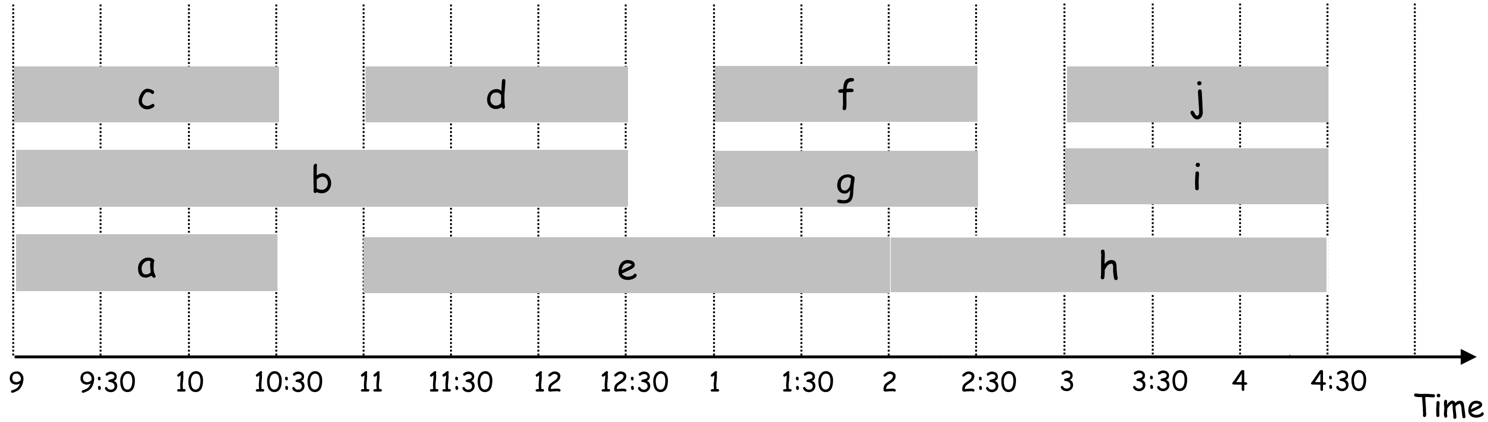


Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses only 3 classrooms.



Greedy template.

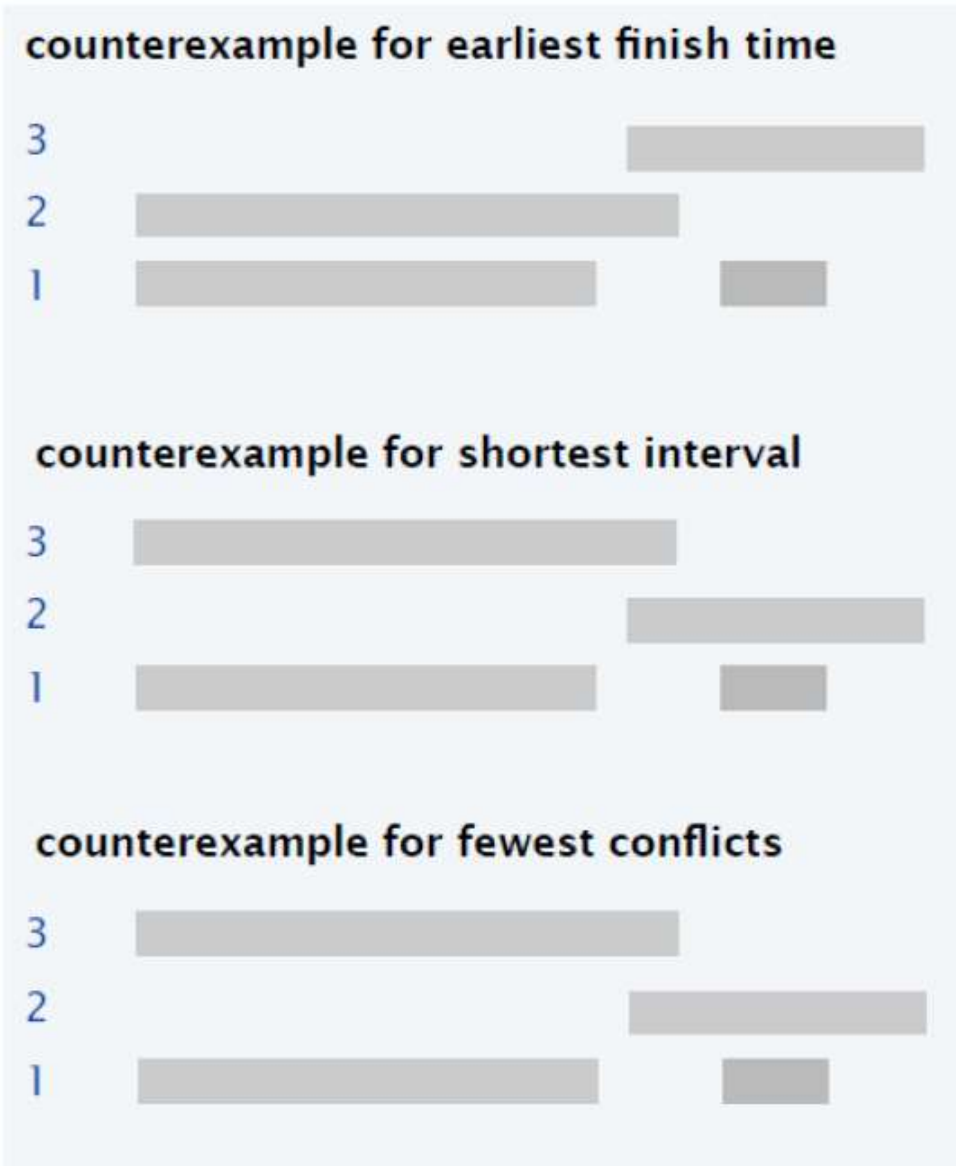
- [Earliest finish time]
  Consider jobs in ascending order of finish time f_j.

- [Shortest interval]
  Consider jobs in ascending order of interval length $f_j - s_j$.

- [Fewest conflicts]
  For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j.

- [Earliest start time]
  Consider jobs in ascending order of start time s_j.

Interval Partitioning



Interval Partitioning: Lower Bound on Optimal Solution

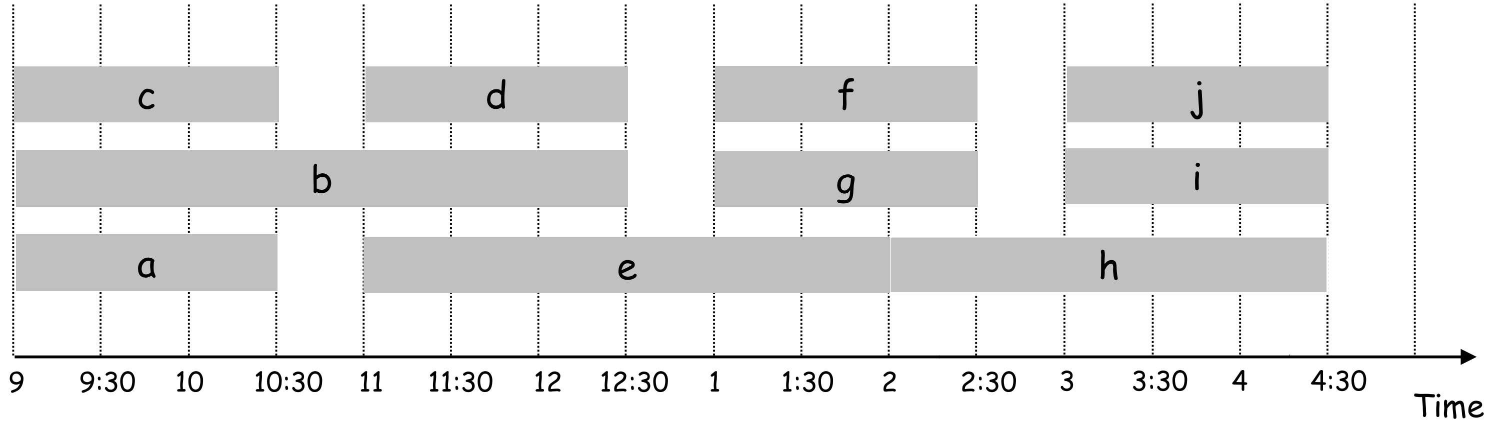
Def. The depth of a set of open intervals is the maximum number of intervals that contain any given time.

Key observation. Number of classrooms needed $\geq$ depth.

Ex: Depth of schedule below = 3 $\Rightarrow$ schedule below is optimal.

a, b, c all contain 9:30

Q. Does there always exist a schedule equal to the depth of intervals?



Interval Partitioning

EARLIESTSTARTTIMEFIRST(n, s_1, s_2, …, s_n, f_1, f_2, …, f_n)

SORT lectures by start time so that $s_1 \leq s_2 \leq \ldots \leq s_n$.

$d \leftarrow 0$ ← number of allocated classrooms

FOR $j = 1$ TO n

  IF lecture j is compatible with some classroom

    Schedule lecture j in any such classroom k.

  ELSE

    Allocate a new classroom $d + 1$.

    Schedule lecture j in classroom $d + 1$.

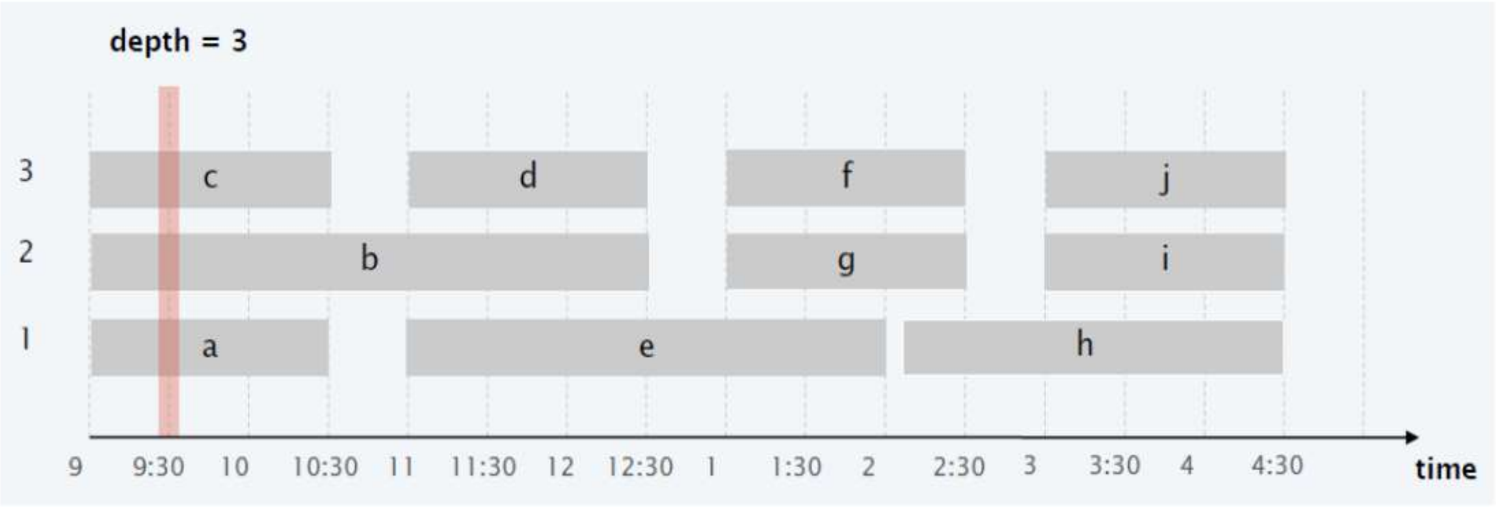
    $d \leftarrow d + 1$

RETURN schedule.

Interval Partitioning

Proof of optimality (lower bound)

- #classrooms needed ≥ maximum "depth" at any point
  - depth = number of lectures running at that time
- We now show that our greedy algorithm uses only these many classrooms!



Interval Partitioning: Greedy Analysis

Observation. Greedy algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Greedy algorithm is optimal.

Pf.

- Let d = number of classrooms that the greedy algorithm allocates.
- Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all d-1 other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.
- Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
- Key observation $\Rightarrow$ all schedules use $\geq$ d classrooms. ▪

Scheduling to Minimize Lateness

We have a single resource and a set of n requests to use the resource for an interval of time.

Assume that the resource is available starting at time s.

Each request is now more flexible. Instead of a start time and finish time, the request i has a deadline d_i, and it requires a contiguous time interval of length t_i, but it is willing to be scheduled at any time before the deadline.

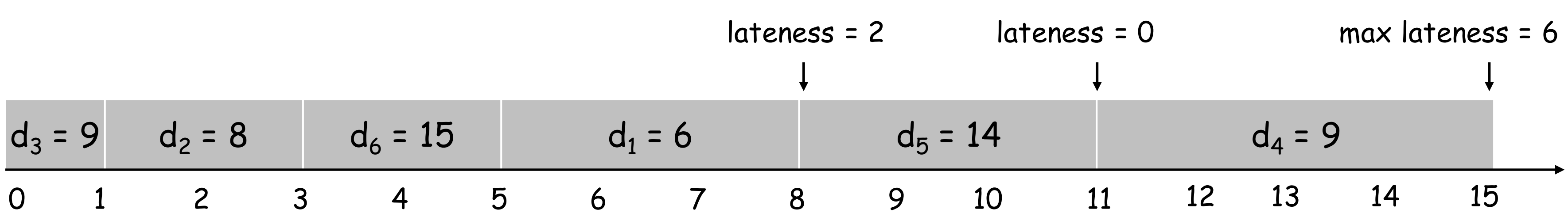
Scheduling to Minimize Lateness

Minimizing lateness problem.

- Single resource processes one job at a time.
- Job j requires t_j units of processing time and is due at time d_j.
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness: $\ell_j = \max\{0, f_j - d_j\}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

- [Earliest deadline first] Consider jobs in ascending order of deadline d_j.

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

counterexample

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

	1	2
t_j	1	10
d_j	2	10

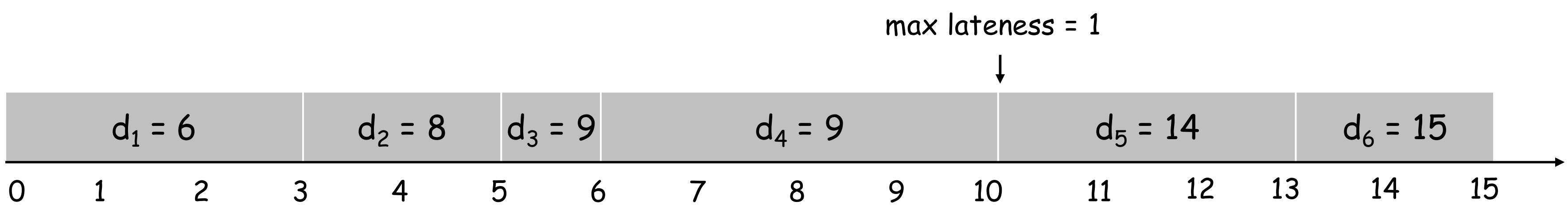
counterexample

Minimizing Lateness: Greedy Algorithm

Greedy algorithm. Earliest deadline first.

```
Sort n jobs by deadline so that d1 ≤ d2 ≤ … ≤ dn

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + tj]
   sj ← t, fj ← t + tj
   t ← t + tj
output intervals [sj, fj]
```



The knapsack problem

- n objects, each with a weight $w_i > 0$
  a profit $p_i > 0$
  capacity of knapsack: M

Maximize $\sum_{1 \le i \le n} p_i x_i$

Subject to $\sum_{1 \le i \le n} w_i x_i \le M$

$0 \le x_i \le 1, \ 1 \le i \le n$

The knapsack algorithm

- The greedy algorithm:
  Step 1: Sort p_i/w_i into nonincreasing order.
  Step 2: Put the objects into the knapsack according to the sorted sequence as possible as we can.
- e. g.
  $n = 3$, $M = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$
  $(w_1, w_2, w_3) = (18, 15, 10)$
  Sol: $p_1/w_1 = 25/18 = 1.39$
  $p_2/w_2 = 24/15 = 1.6$
  $p_3/w_3 = 15/10 = 1.5$
  Optimal solution: $x_1 = 0$, $x_2 = 1$, $x_3 = 1/2$
  total profit $= 24 + 7.5 = 31.5$

The 2-way merging problem

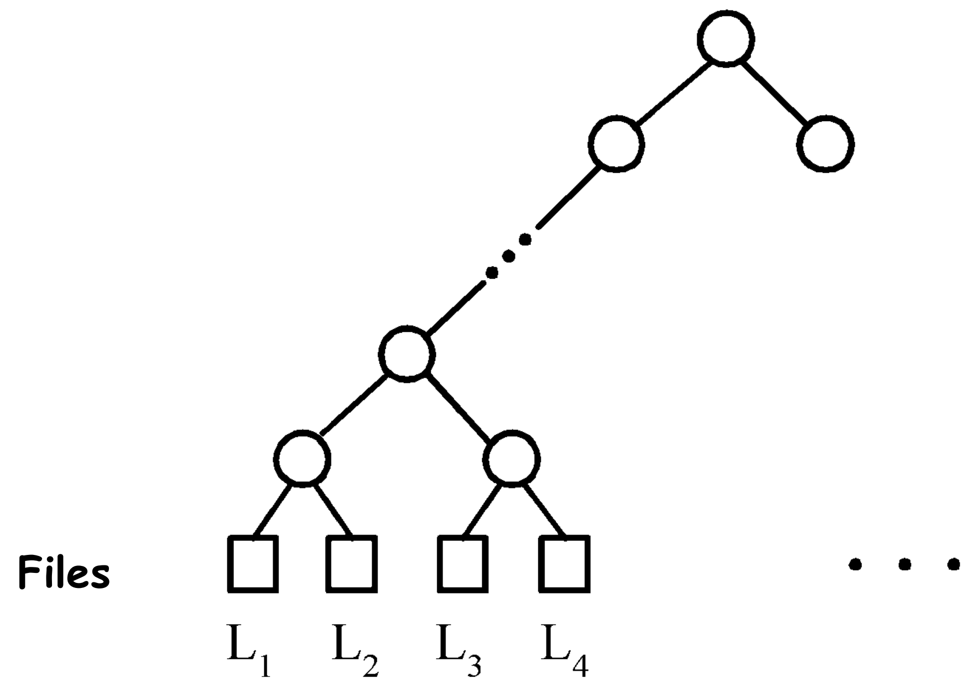
- # of comparisons required for the linear 2-way merge algorithm (Each merge step involved margining of two files) is $m_1 + m_2 - 1$ where m_1 and m_2 are the lengths of the two sorted lists respectively.
  - 2-way merging example

    2 3 5 6

    1 4 7 8
- The problem: There are n sorted lists, each of length m_i. What is the optimal sequence of merging process to merge these n lists into one sorted list ?

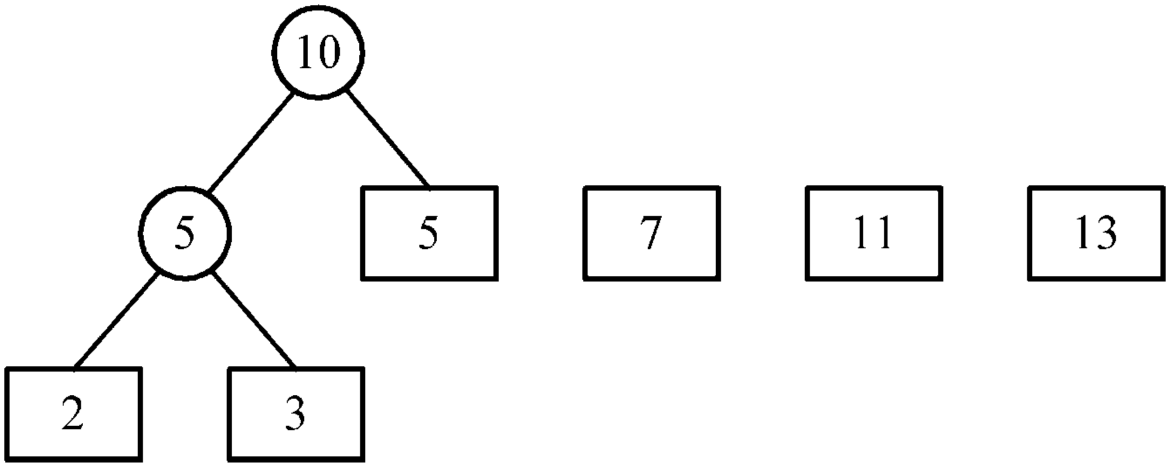
Extended binary trees

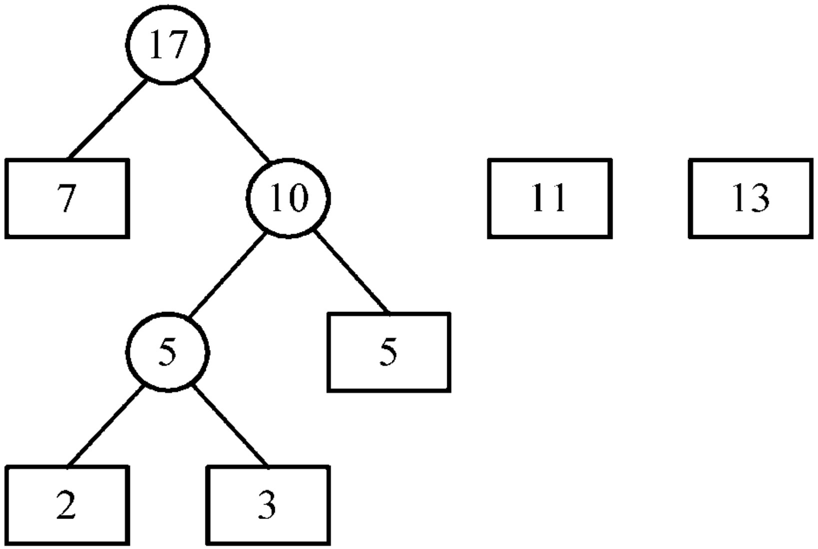
- Two-way merge pattern can be represented by a binary merge pattern tree

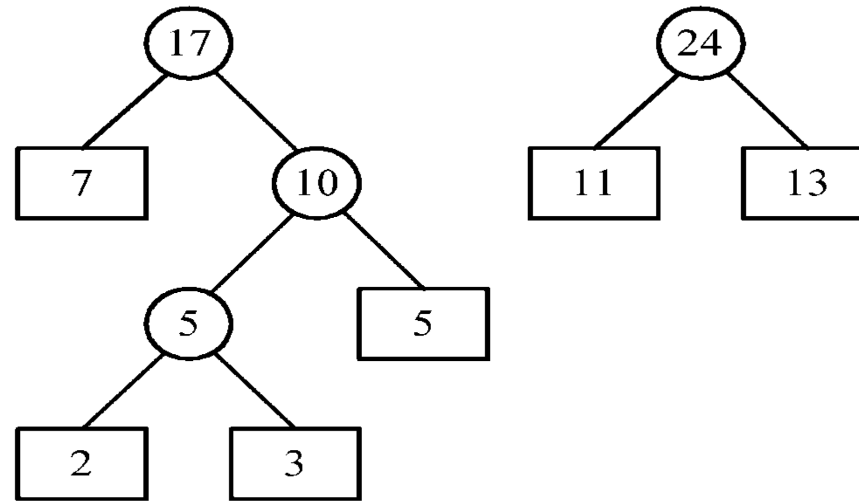


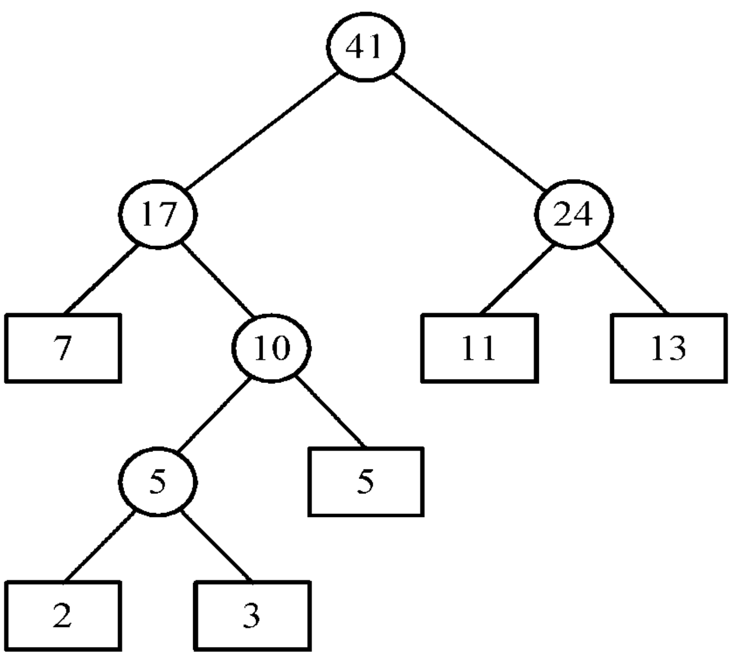
An example of 2-way merging

- Example: 6 sorted lists with lengths 2, 3, 5, 7, 11 and 13.









- Time complexity for generating an optimal extended binary tree:O(n log n)
- Using min-heap

Huffman codes

- In telecommunication, how do we represent a set of messages, each with an access frequency, by a sequence of 0's and 1's?
- To minimize the transmission and decoding costs, we may use short strings to represent more frequently used messages.
- This problem can by solved by using an extended binary tree which is used in the 2-way merging problem.

Huffman Code Problem

- Huffman's algorithm achieves data compression by finding the best variable length binary encoding scheme for the symbols that occur in the file to be compressed.

Huffman Code Problem

- The more frequent a symbol occurs, the shorter should be the Huffman binary word representing it.

- The Huffman code is a prefix-free code.
  - No prefix of a code word is equal to another codeword.

Overview

- Huffman codes: compressing data (savings of 20% to 90%)
- Huffman's greedy algorithm uses a table of the frequencies of occurrence of each character to build up an optimal way of representing each character as a binary string

C: Alphabet

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 16.3 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. If each character is assigned a 3-bit codeword, the file can be encoded in 300,000 bits. Using the variable-length code shown, the file can be encoded in 224,000 bits.

Example

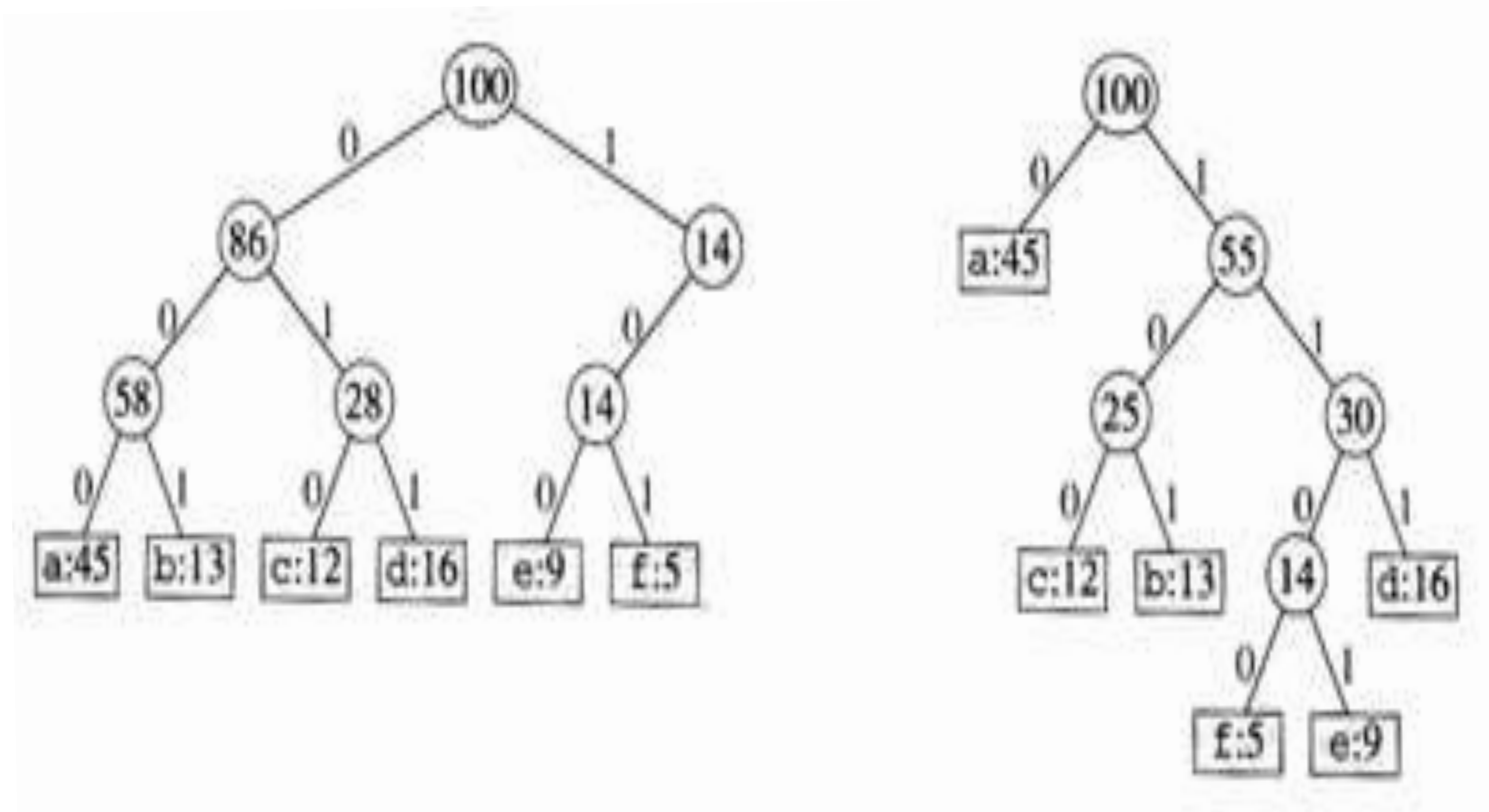
- Assume we are given a data file that contains only 6 symbols, namely a, b, c, d, e, f With the following frequency table:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- Find a variable length prefix-free encoding scheme that compresses this data file as much as possible?

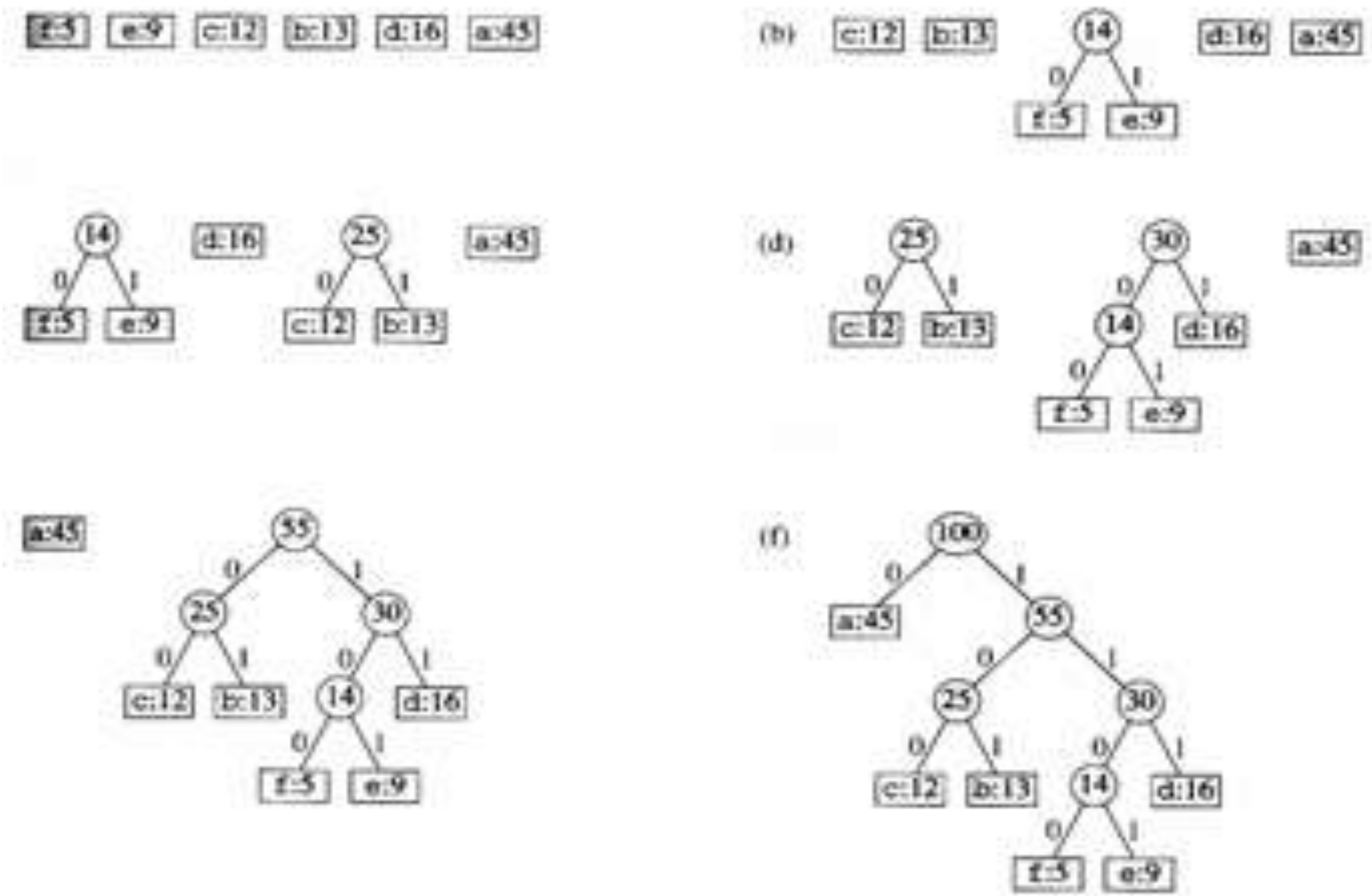
Huffman Code Problem

- Left tree represents a fixed length encoding scheme
- Right tree represents a Huffman encoding scheme



(a) (b)

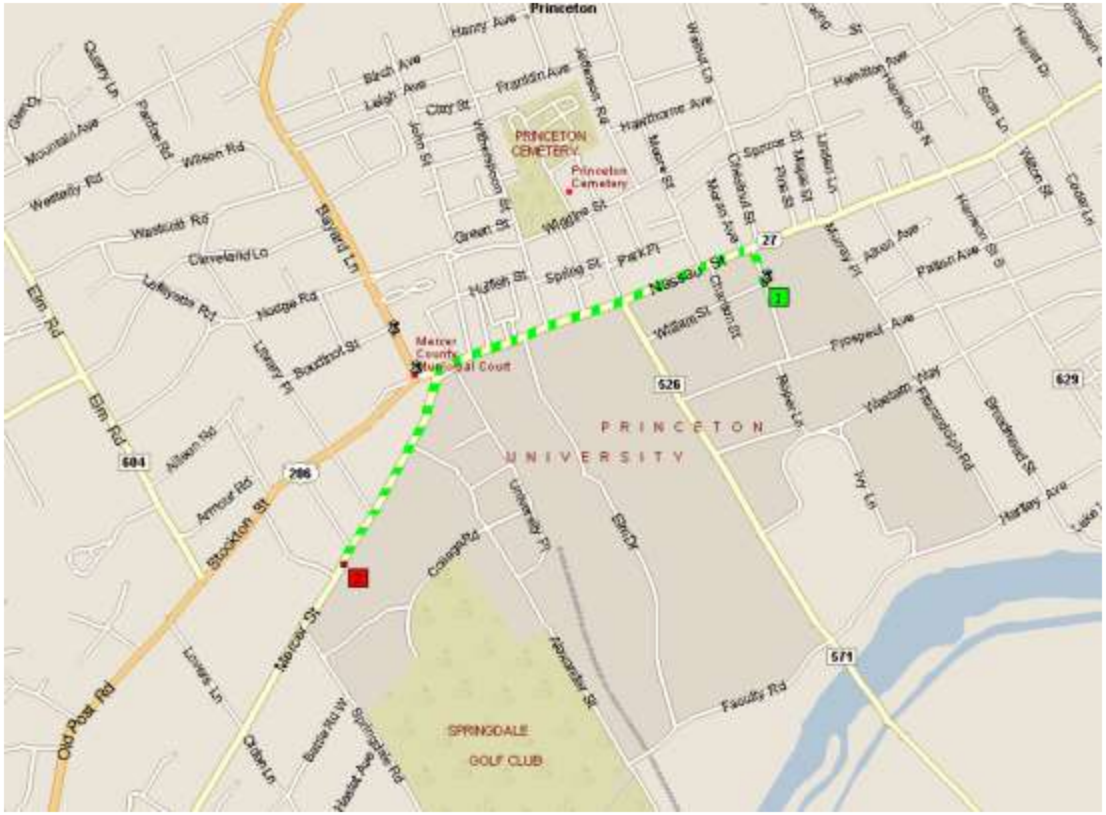
Example



Constructing A Huffman Code

```
H = new Heap()
for each w_i
        T = new Tree(w_i)
        H.Insert(T)
while H.Size() > 1
        T_1 = H.DeleteMin()
        T_2 = H.DeleteMin()
        T_3 = Merge(T_1, T_2)
        H.Insert(T_3)
```

4.4 Shortest Paths in a Graph



shortest path from Princeton CS department to Einstein's house

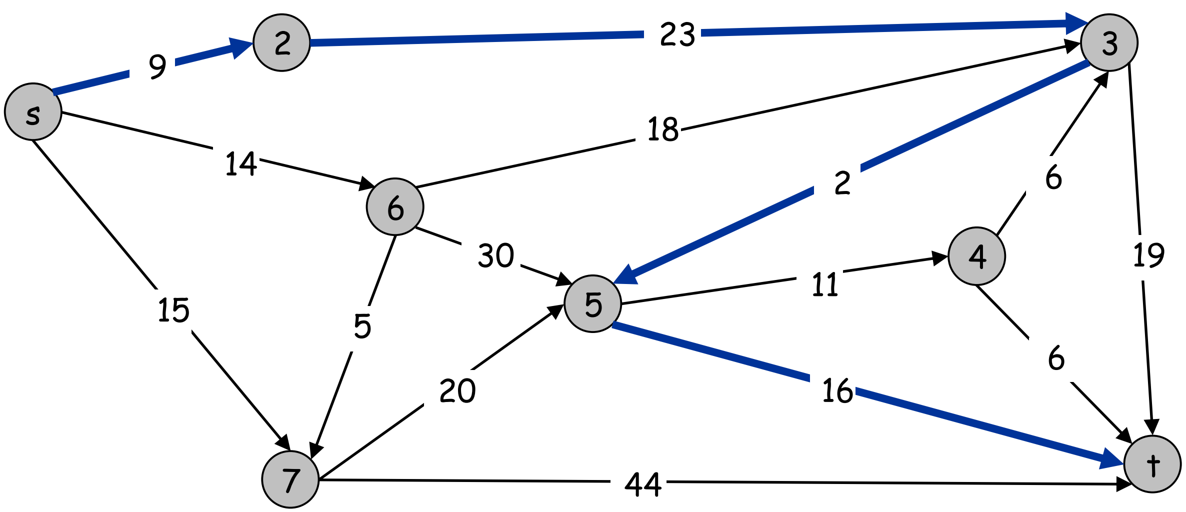
Shortest Path Problem

Shortest path network.

- Directed graph $G = (V, E)$.
- Source s, destination t.
- Length ℓ_e = length of edge e.

Shortest path problem: find shortest directed path from s to t.

cost of path = sum of edge costs in path



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 48.

Dijkstra's Algorithm

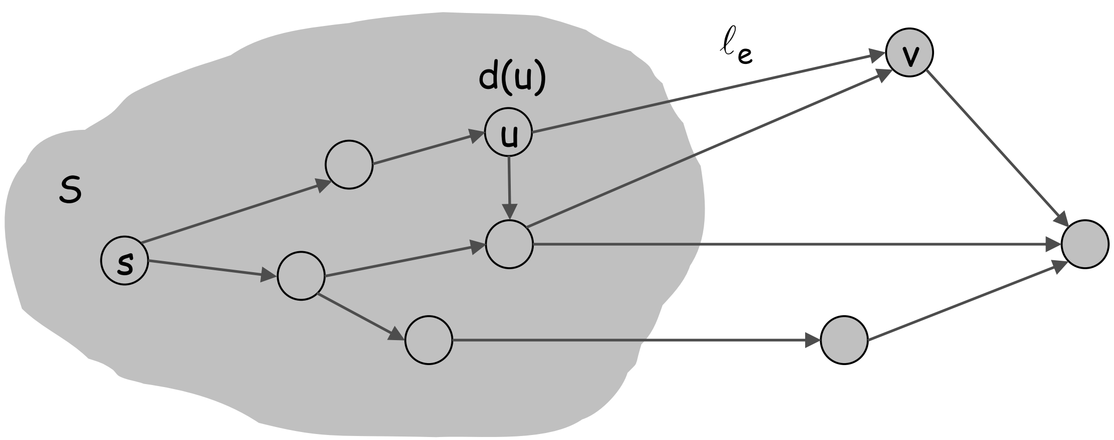
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v (i.e., v $\notin$ S) that minimizes

$$\pi(v) = \min_{e = (u,v) \, : \, u \in S} d(u) + \ell_e ,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



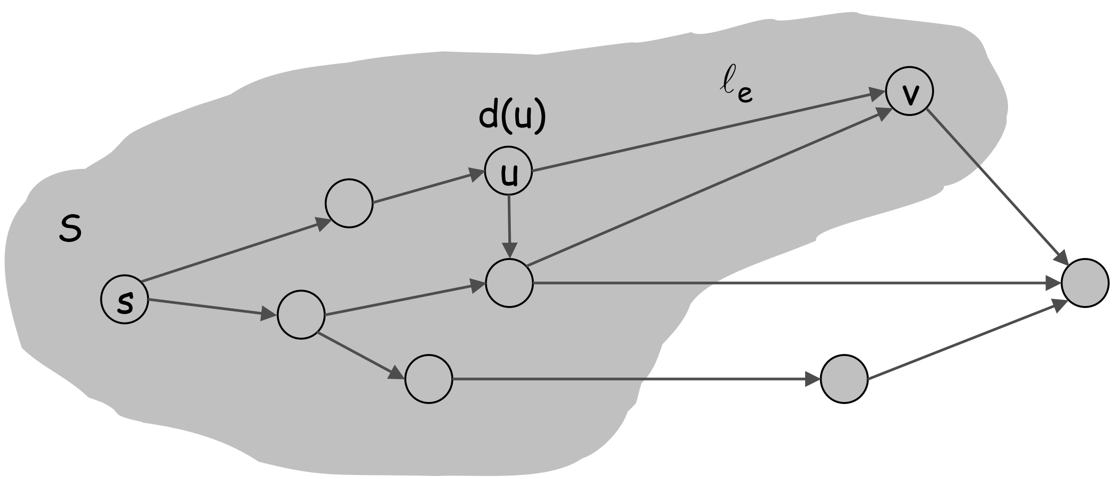
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

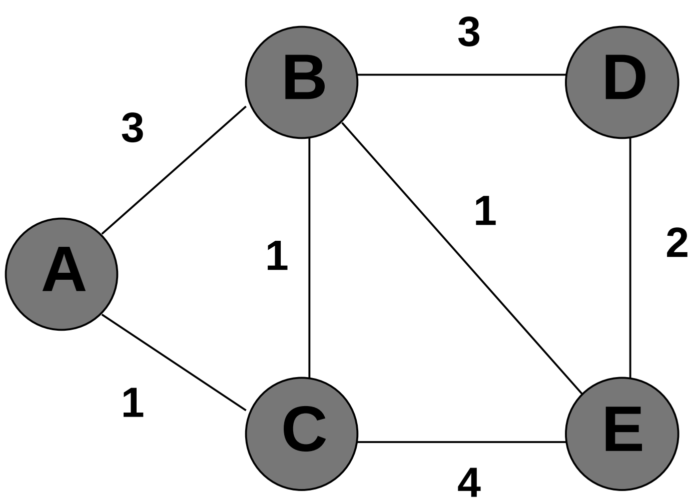
$$\pi(v) = \min_{e = (u,v)\,:\, u \in S} d(u) + \ell_e ,$$

shortest path to some u in explored part, followed by a single edge (u, v)

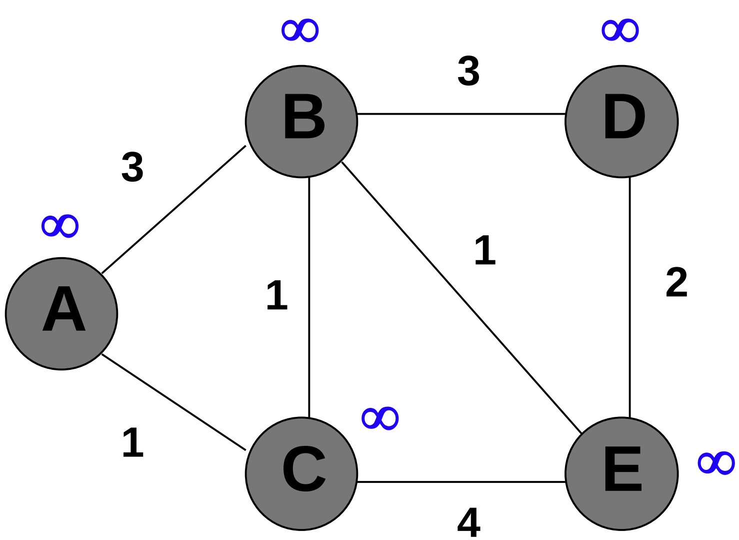
add v to S, and set d(v) = π(v).



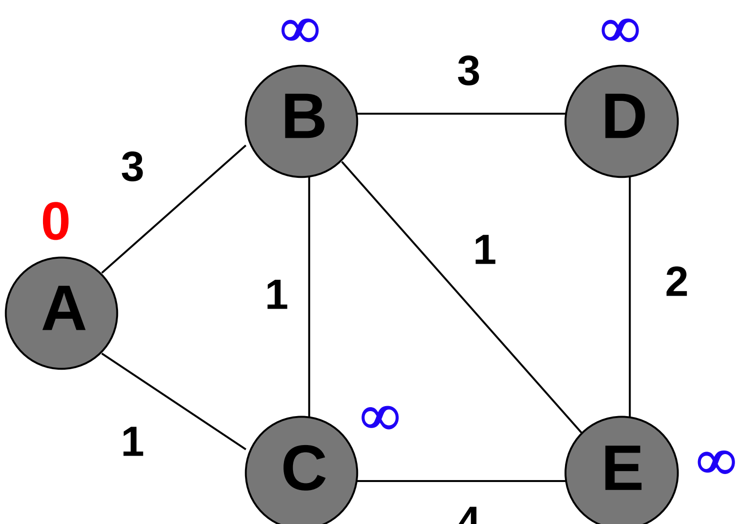
```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```



```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

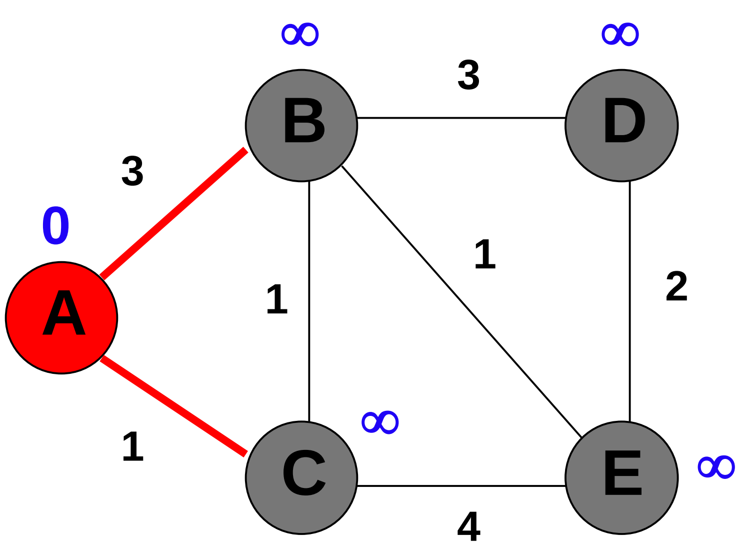


Heap

A	**0**
B	∞
C	∞
D	∞
E	∞

DIJKSTRA(G, s)

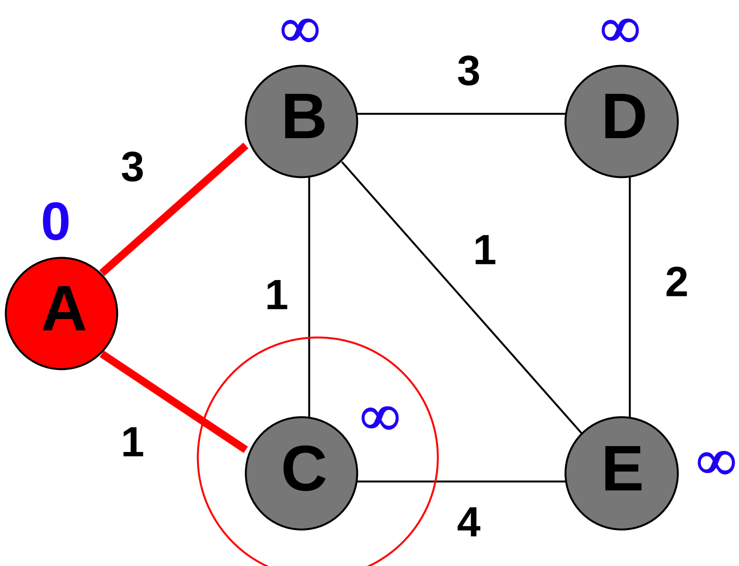
```
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B	∞
C	∞
D	∞
E	∞

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```



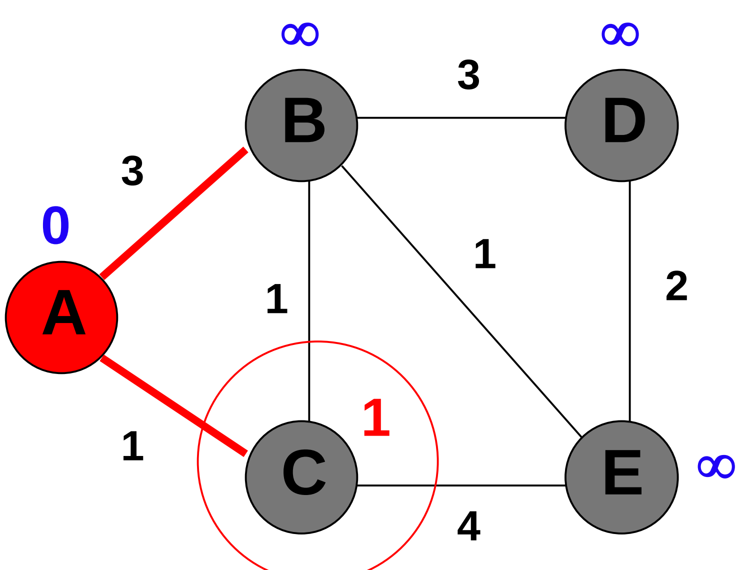
Heap

B	∞
C	∞
D	∞
E	∞

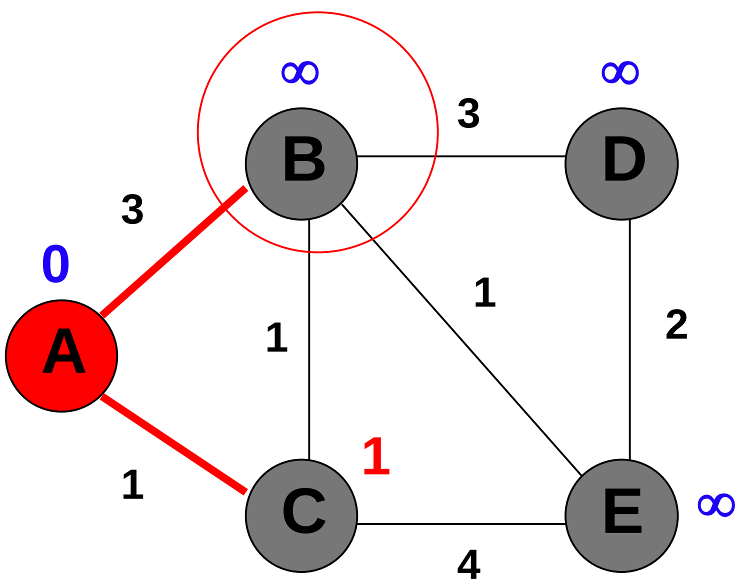
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Heap

C 1
B ∞
D ∞
E ∞



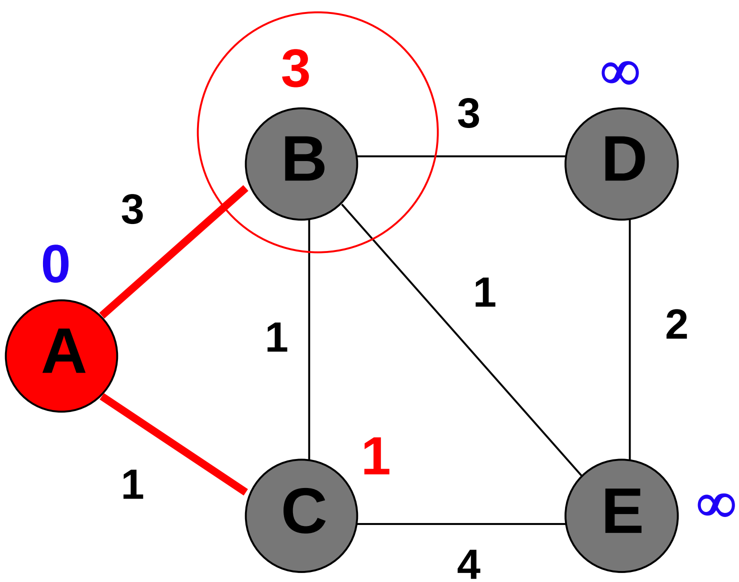
```
DIJKSTRA(G, s)
for all v ∈ V
    dist[v] ← ∞
    prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
    u ← EXTRACTMIN(Q)
    for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
            dist[v] ← dist[u] + w(u, v)
            DECREASEKEY(Q, v, dist[v])
            prev[v] ← u
```



Heap

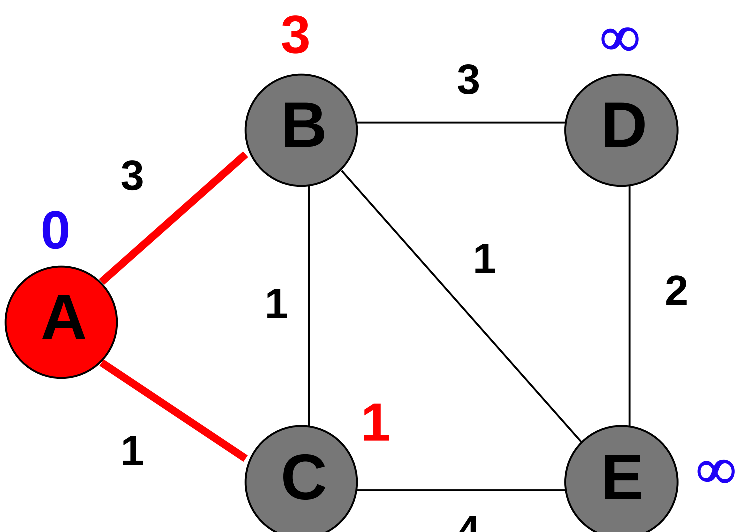
C	1
B	∞
D	∞
E	∞

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```



Heap	
C	1
B	3
D	∞
E	∞

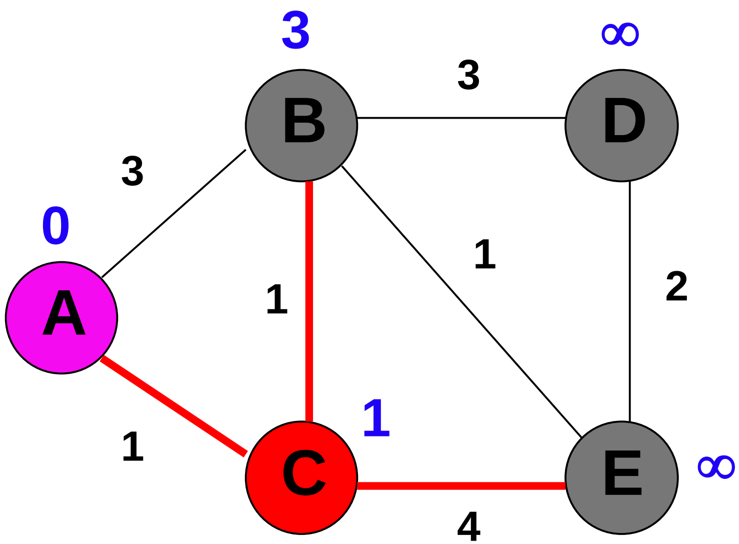
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

C	1
B	3
D	∞
E	∞

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



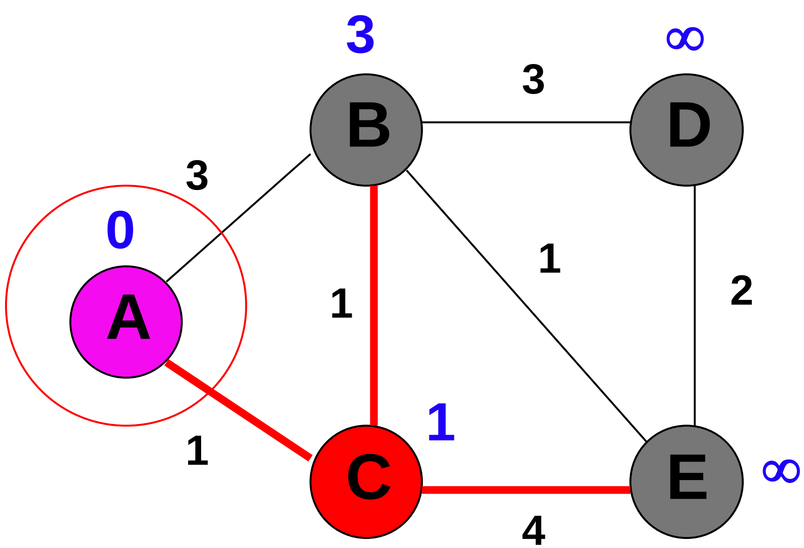
Heap

B 3
D ∞
E ∞

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Heap

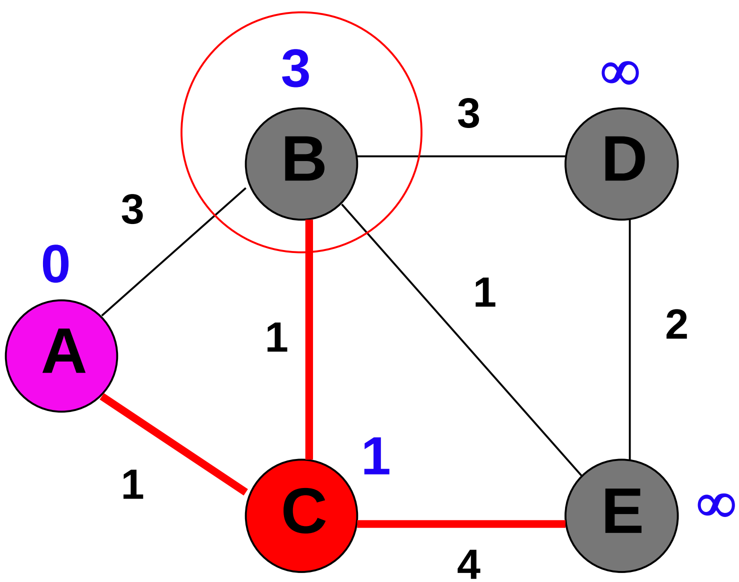
B	3
D	∞
E	∞



```
Dijkstra(G, s)
for all v ∈ V
    dist[v] ← ∞
    prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
    u ← ExtractMin(Q)
    for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
            dist[v] ← dist[u] + w(u, v)
            DecreaseKey(Q, v, dist[v])
            prev[v] ← u
```

Heap

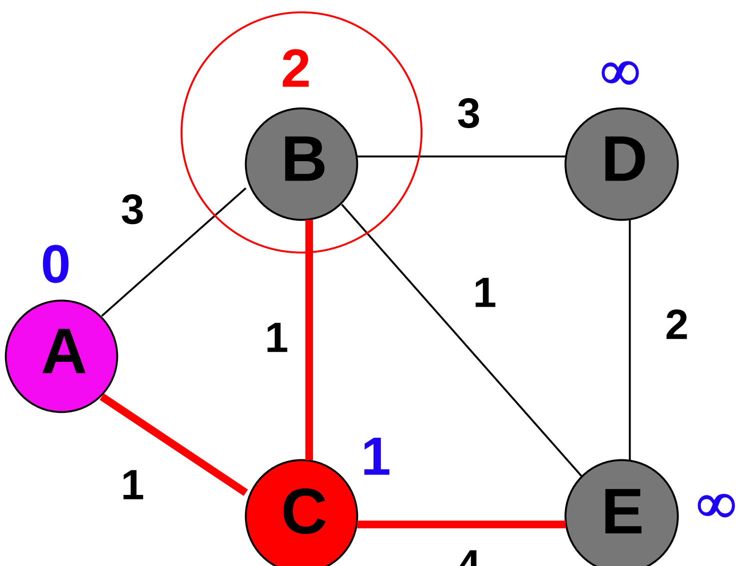
B	3
D	∞
E	∞



```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

Heap

B 2
D ∞
E ∞

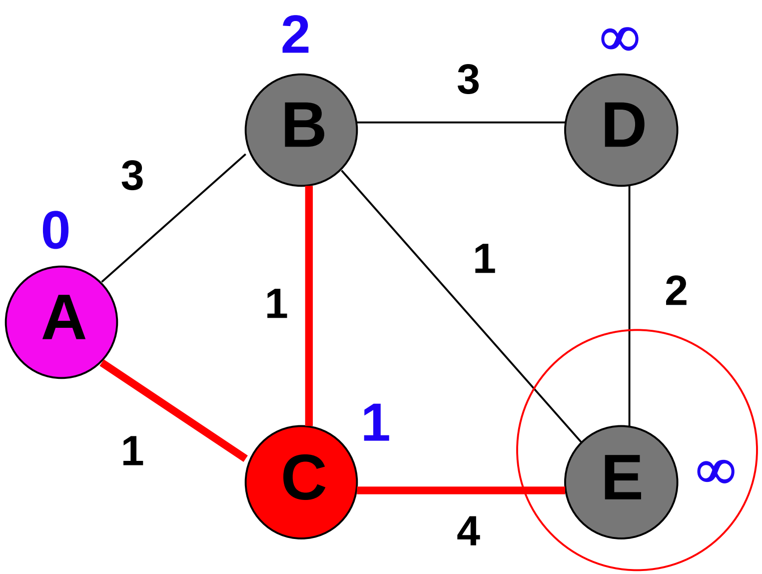


DIJKSTRA(G, s)

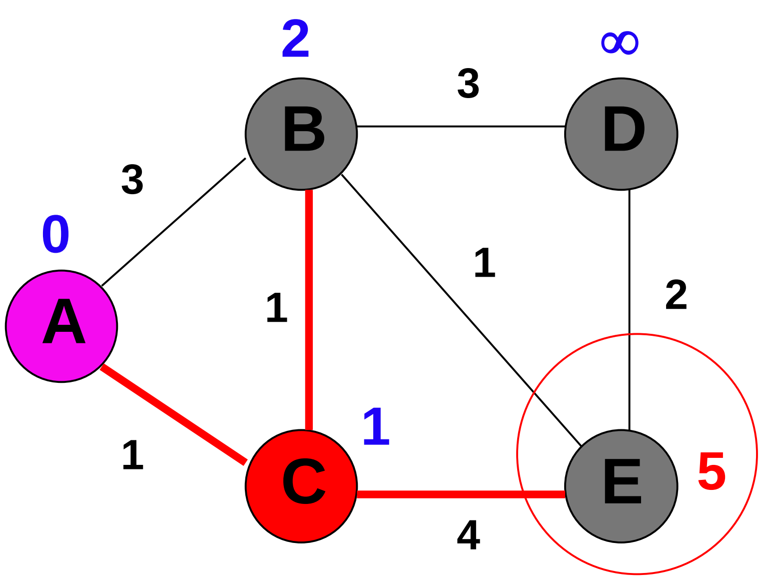
```
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

Heap

B	2
D	∞
E	∞



```
DIJKSTRA(G, s)
for all v ∈ V
    dist[v] ← ∞
    prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
    u ← EXTRACTMIN(Q)
    for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
            dist[v] ← dist[u] + w(u, v)
            DECREASEKEY(Q, v, dist[v])
            prev[v] ← u
```



Heap

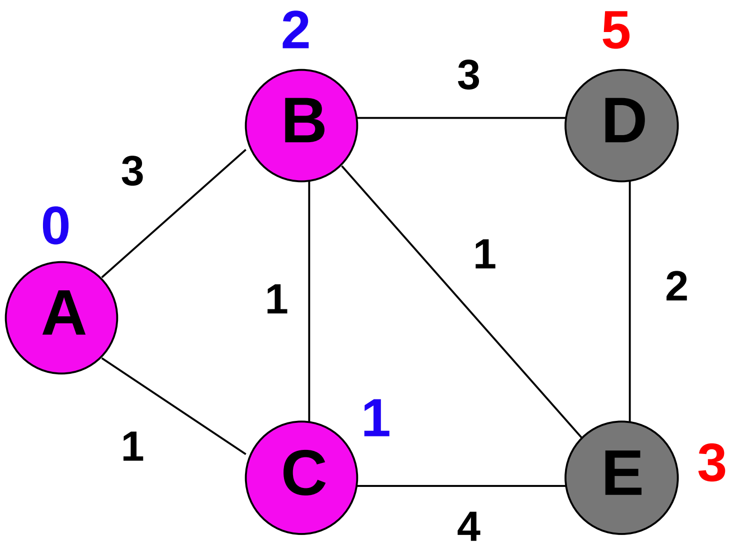
B	2
E	5
D	∞

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Heap

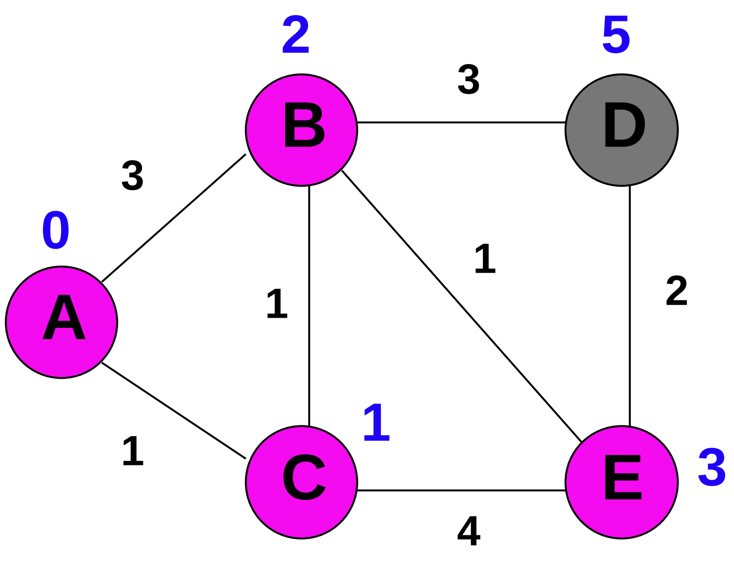
E 3

D 5



```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

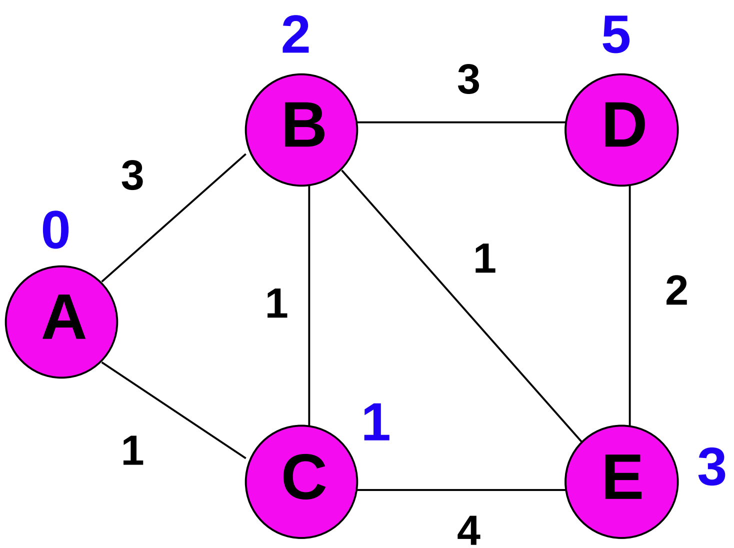
Heap	
D	5



Dijkstra(G, s)

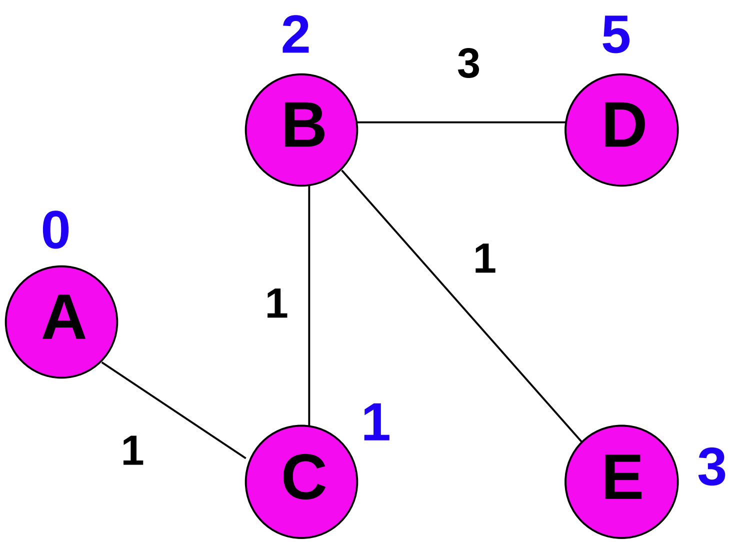
```
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

Heap



```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Heap



Running time?

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Running time?

DIJKSTRA(G, s)

```
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

1 call to MakeHeap

Running time?

$\textsc{Dijkstra}(G, s)$

for all $v \in V$
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
$Q \leftarrow \textsc{MakeHeap}(V)$
while $!\textsc{Empty}(Q)$
    $u \leftarrow \textsc{ExtractMin}(Q)$
    for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
            $dist[v] \leftarrow dist[u] + w(u, v)$
            $\textsc{DecreaseKey}(Q, v, dist[v])$
            $prev[v] \leftarrow u$

|V| iterations

Running time?

$\textsc{Dijkstra}(G, s)$

for all $v \in V$
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
$Q \leftarrow \textsc{MakeHeap}(V)$
while $!\textsc{Empty}(Q)$
    $u \leftarrow \textsc{ExtractMin}(Q)$ **|V| calls**
    for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
            $dist[v] \leftarrow dist[u] + w(u, v)$
            $\textsc{DecreaseKey}(Q, v, dist[v])$
            $prev[v] \leftarrow u$

Running time?

DIJKSTRA(G, s)

for all $v \in V$
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
$Q \leftarrow \text{MAKEHEAP}(V)$
while $!\text{EMPTY}(Q)$
    $u \leftarrow \text{EXTRACTMIN}(Q)$
    for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
            $dist[v] \leftarrow dist[u] + w(u, v)$
            $\boxed{\text{DECREASEKEY}(Q, v, dist[v])}$ **O(|E|) calls**
            $prev[v] \leftarrow u$

Running time?

Depends on the heap implementation

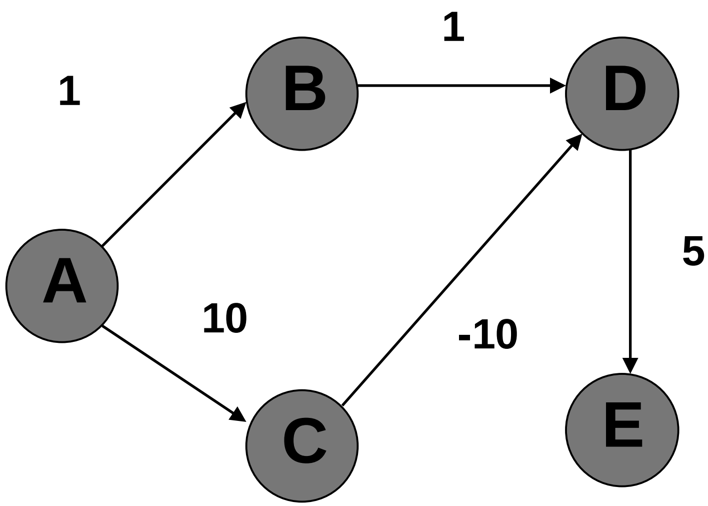
	1 MakeHeap	$\lvert V\rvert$ ExtractMin	$\lvert E\rvert$ DecreaseKey	Total
Array	$O(\lvert V\rvert)$	$O(\lvert V\rvert^2)$	$O(\lvert E\rvert)$	$O(\lvert V\rvert^2)$
Bin heap	$O(\lvert V\rvert)$	$O(\lvert V\rvert \log \lvert V\rvert)$	$O(\lvert E\rvert \log \lvert V\rvert)$	$O((\lvert V\rvert+\lvert E\rvert) \log \lvert V\rvert)$
				$O(\lvert E\rvert \log \lvert V\rvert)$

Running time?

Depends on the heap implementation

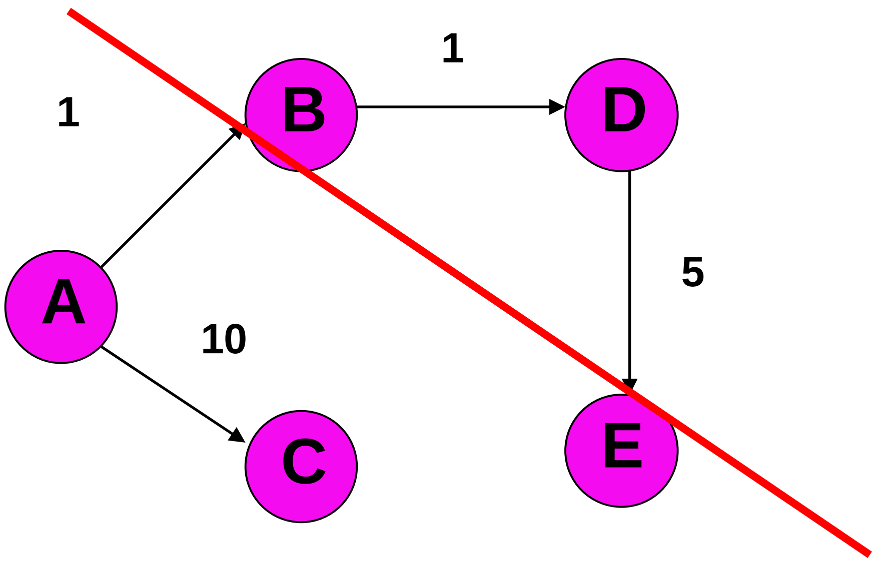
	1 MakeHeap	|V| ExtractMin	|E| DecreaseKey	Total
Array	O(|V|)	$O(\|V\|^2)$	O(|E|)	$O(\|V\|^2)$
Bin heap	O(|V|)	O(|V| log |V|)	O(|E| log |V|)	O((|V|+|E|) log |V|) O(|E| log |V|)

What about Dijkstra's on…?



What about Dijkstra's on…?

Dijkstra's algorithm only works for positive edge weights



Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.

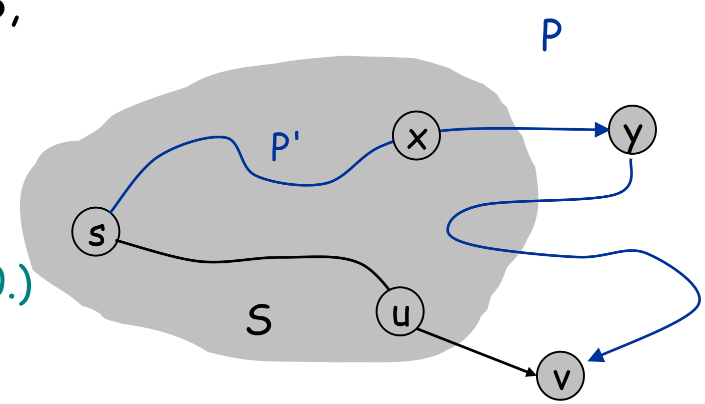
Pf. (by induction on $|S|$)

Base case: $|S| = 1$ is trivial.

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be the next node added to S, and let u-v be the chosen edge.
- The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
- Consider any s-v path P. Show that it is no shorter than $\pi(v)$.
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S and path y-v has positive weight.

(Boundary Case: P has same length with len(y-v) = 0.)

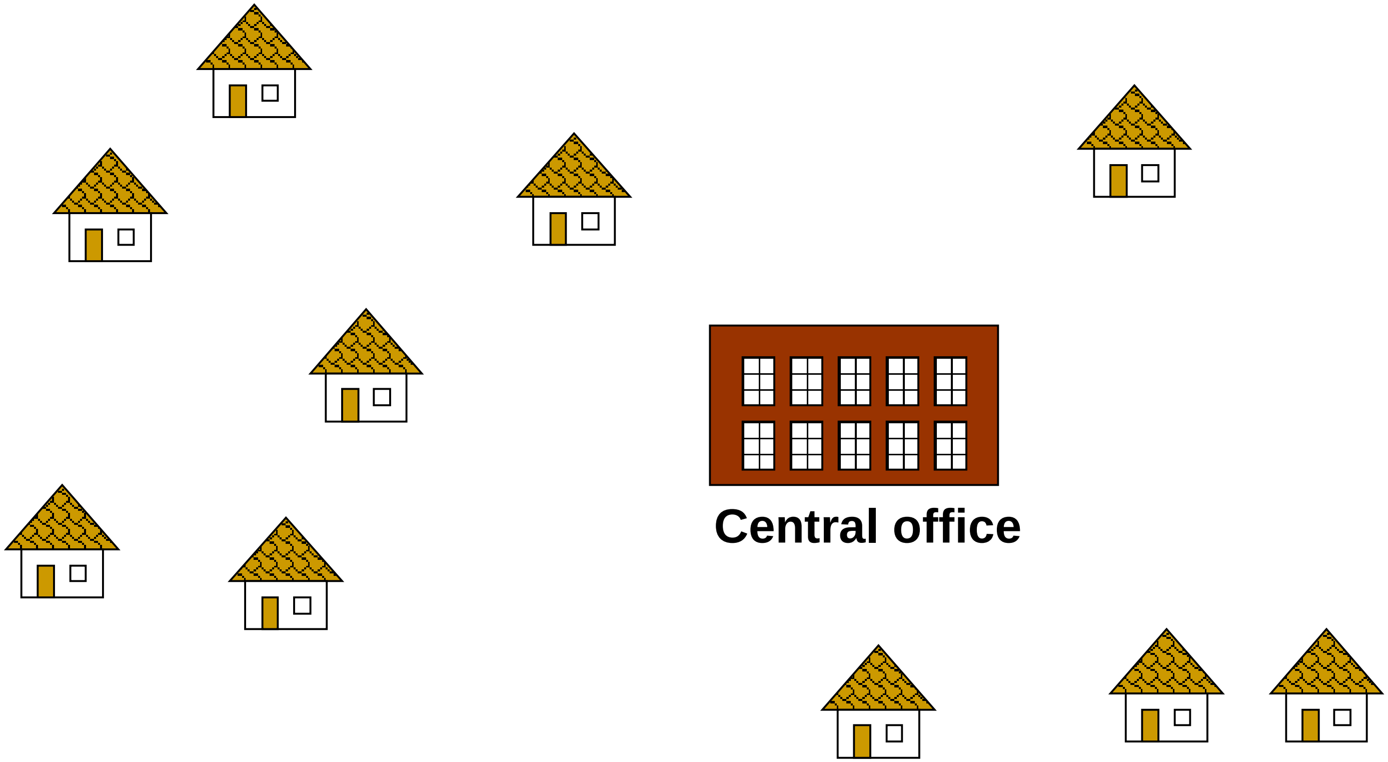


$$\ell(P) \geq \ell(P') + \ell(x,y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v)$$

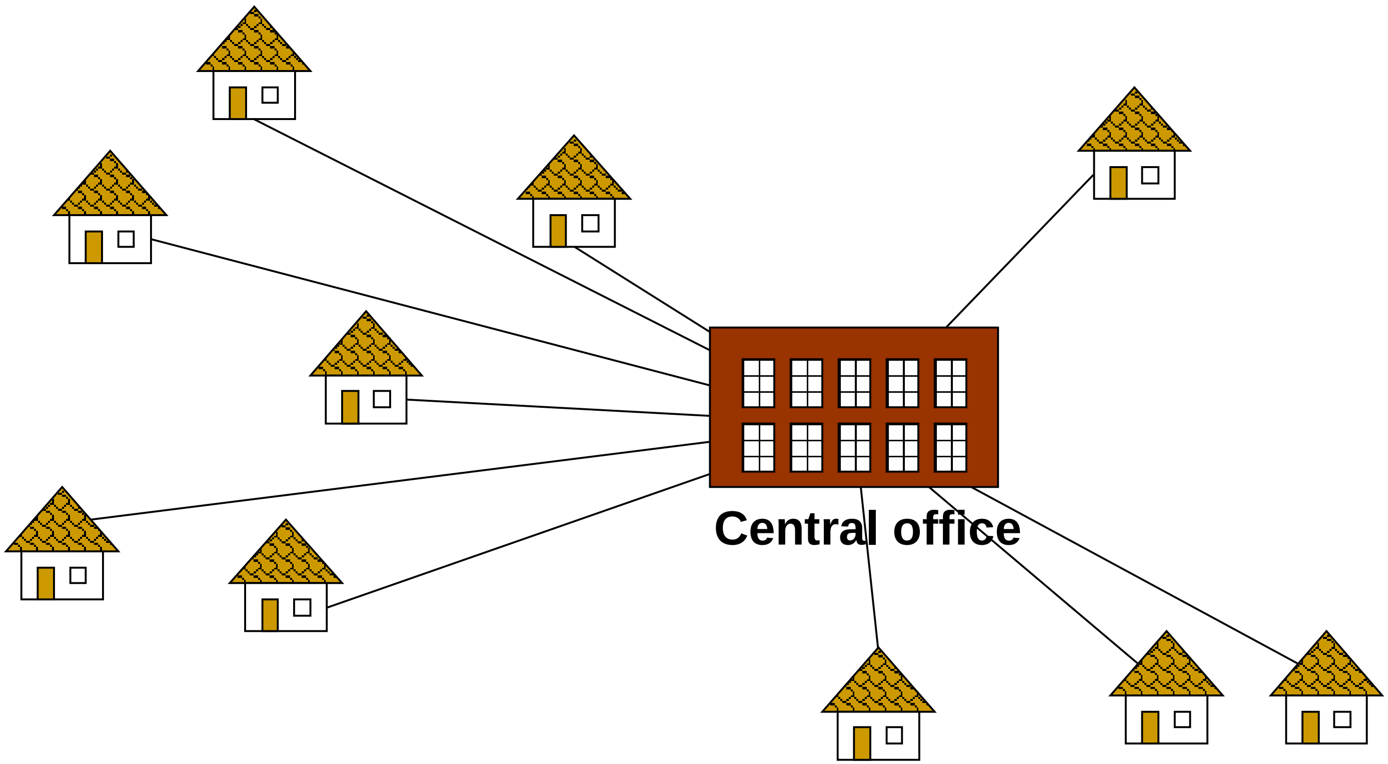
nonnegative weights (lengths) — inductive hypothesis — defn of $\pi(y)$ — Dijkstra's algorithm chose v instead of y

Minimum Spanning Trees

Problem: Laying Telephone Wire

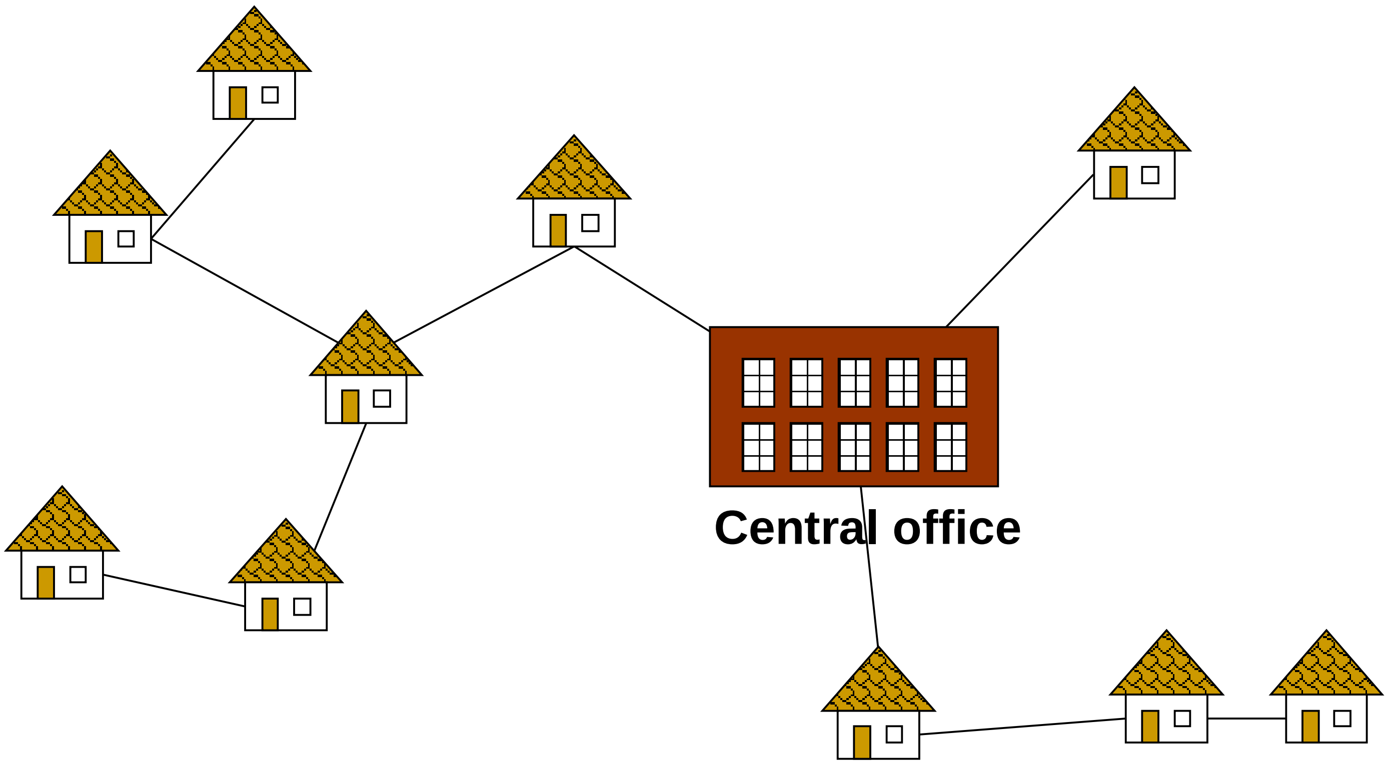


Wiring: Naive Approach



Expensive!

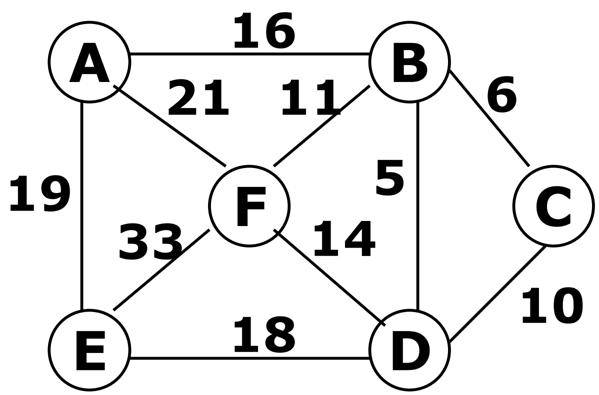
Wiring: Better Approach



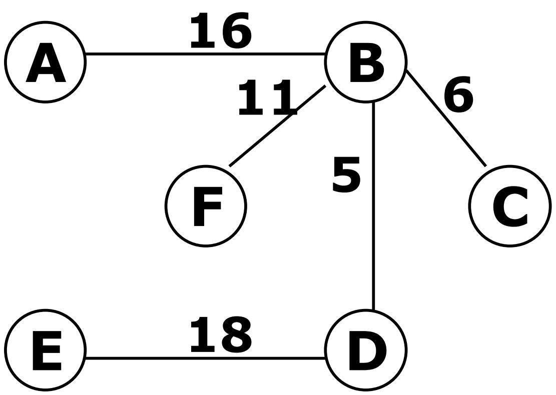
Minimize the total length of wire connecting the customers

Minimum-cost spanning trees

- Suppose you have a connected undirected graph with a weight (or cost) associated with each edge
- The cost of a spanning tree would be the sum of the costs of its edges
- A minimum-cost spanning tree is a spanning tree that has the lowest cost



A connected, undirected graph



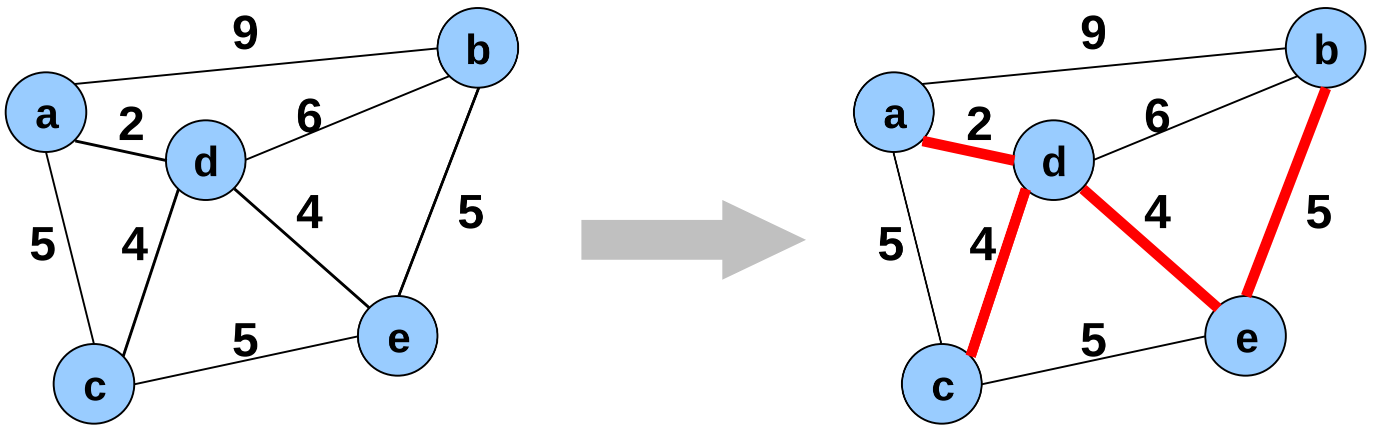
A minimum-cost spanning tree

Minimum Spanning Tree (MST)

A minimum spanning tree is a subgraph of an undirected weighted graph *G*, such that

- **it is a tree (i.e., it is acyclic)**
- **it covers all the vertices *V***
  - **contains $|V| - 1$ edges**
- **the total cost associated with tree edges is the minimum among all possible spanning trees**
- **not necessarily unique**

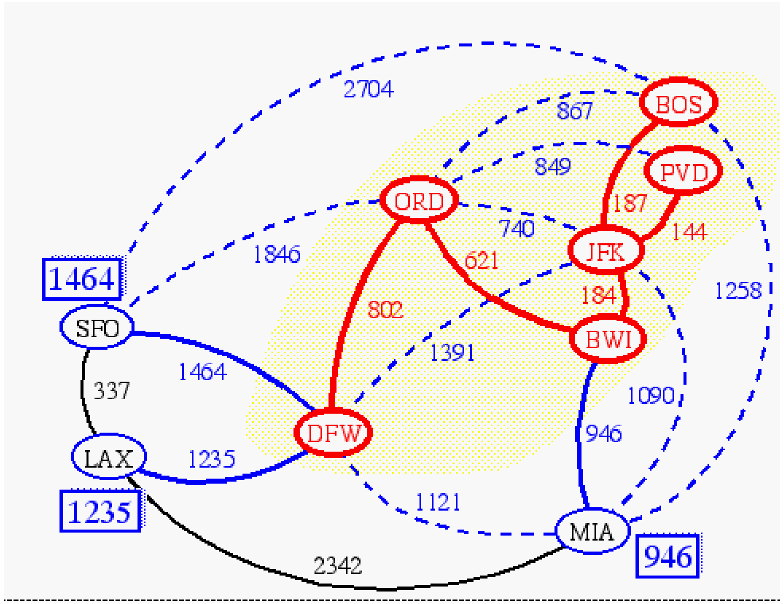
How Can We Generate a MST?



Prim(-Jarnik)'s Algorithm

Similar to Dijkstra's algorithm (for a connected graph)
We pick an arbitrary vertex ***s*** and we grow the MST as a cloud of vertices, starting from ***s***
We store with each vertex ***v*** a label ***d***(***v***) = the smallest weight of an edge connecting ***v*** to a vertex in the cloud

- **At each step:**
  - **We add to the cloud the vertex *u* outside the cloud with the smallest distance label**
  - **We update the labels of the vertices adjacent to *u***



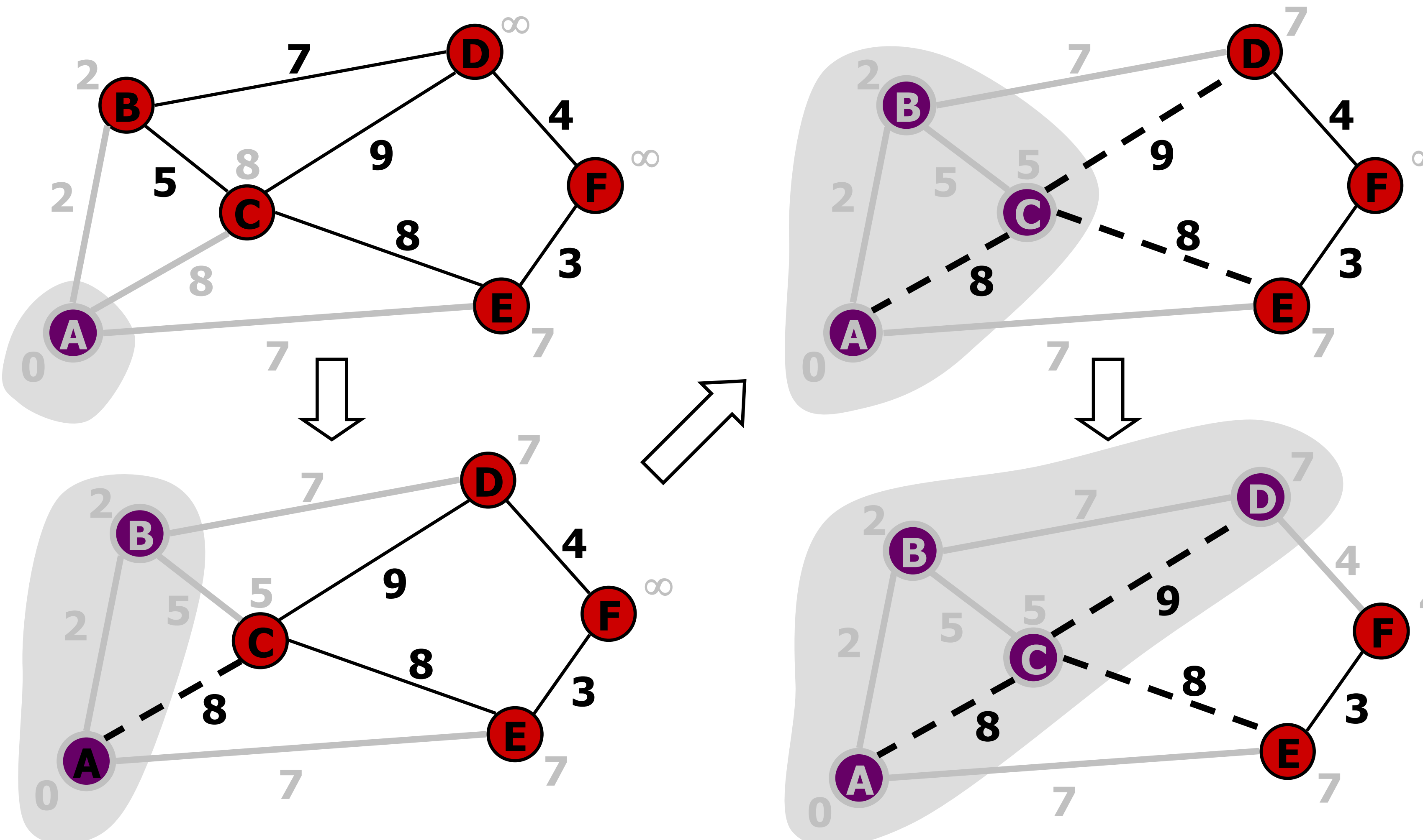
Prim's algorithm

```
T = a spanning tree containing a single node s;
E = set of edges adjacent to s;
while T does not contain all the nodes {
remove an edge (v, w) of lowest cost from E
if w is already in T then discard edge (v, w)
else {
   add edge (v, w) and node w to T
   add to E the edges adjacent to w
}
}
```

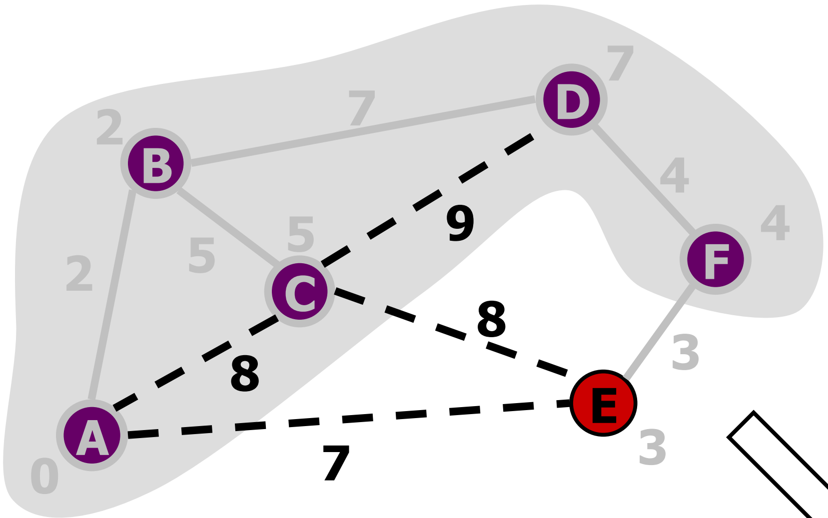
An edge of lowest cost can be found with a priority queue
Testing for a cycle is automatic

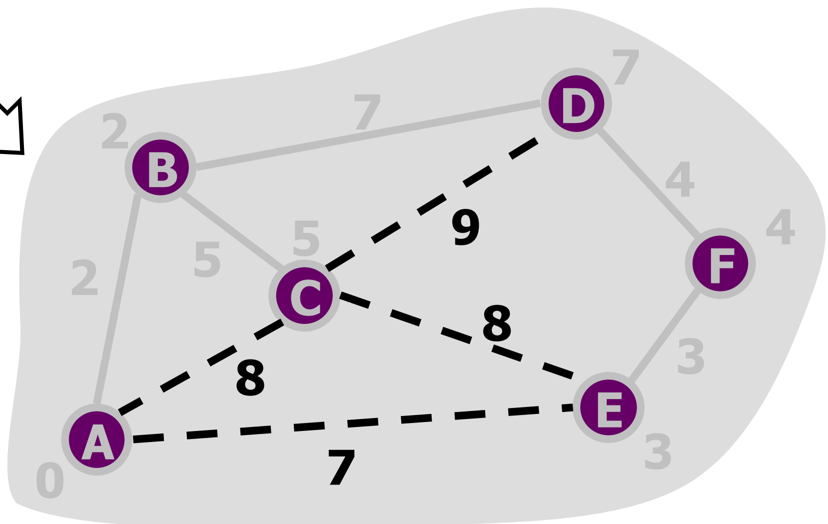
- Hence, Prim's algorithm is far simpler to implement than Kruskal's algorithm (presented below)

Example

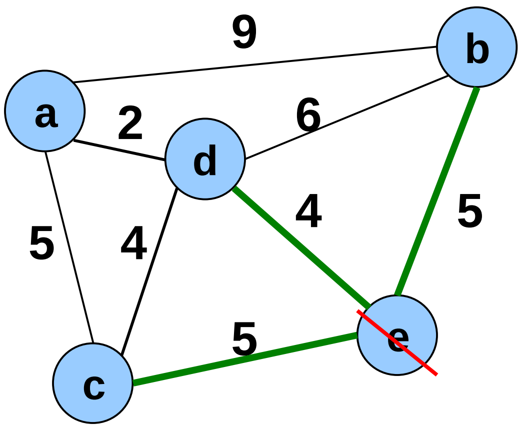


Example (contd.)





Prim's algorithm



e	d	b	c	a
0	∞	∞	∞	∞

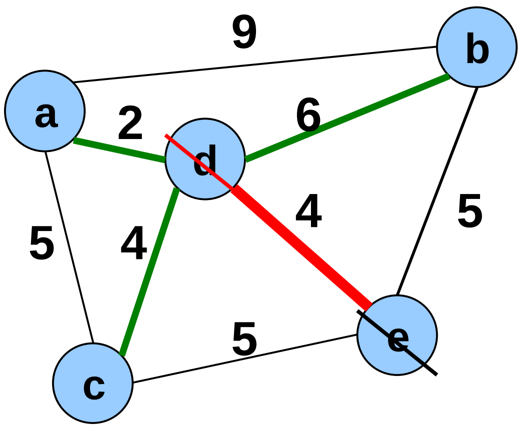
Vertex	Parent
e	-
b	-
c	-
d	-

d	b	c	a
4	5	5	∞

Vertex	Parent
e	-
b	e
c	e
d	e

The MST initially consists of the vertex *e*, and we update the distances and parent for its adjacent vertices

Prim's algorithm



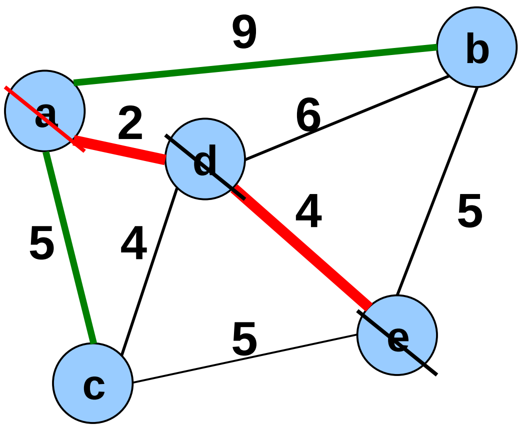
d	b	c	a
4	5	5	∞

Vertex	Parent
e	-
b	e
c	e
d	e

a	c	b
2	4	5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim's algorithm



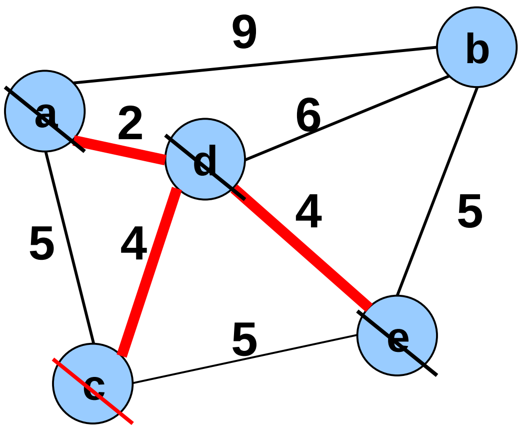
a	c	b
2	4	5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

c	b
4	5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim's algorithm



c	b
4	5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

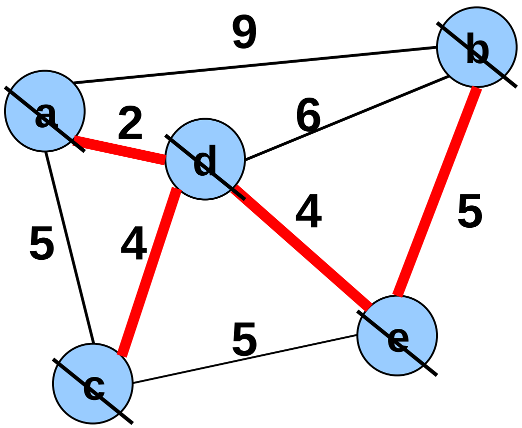
b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Prim's algorithm

b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

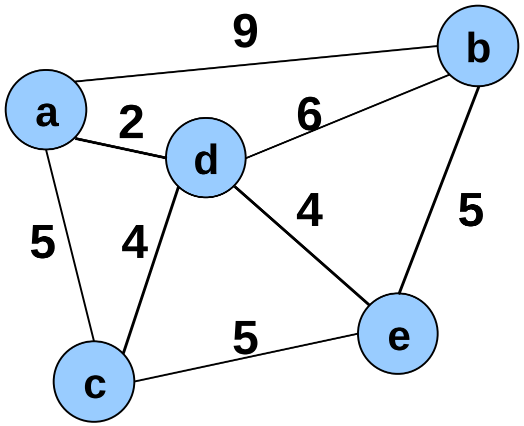


The final minimum spanning tree

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

Another Approach

- **Create a forest of trees from the vertices**
- **Repeatedly merge trees by adding "safe edges" until only one tree remains**
- **A "safe edge" is an edge of minimum weight which does not create a cycle**



forest: {a}, {b}, {c}, {d}, {e}

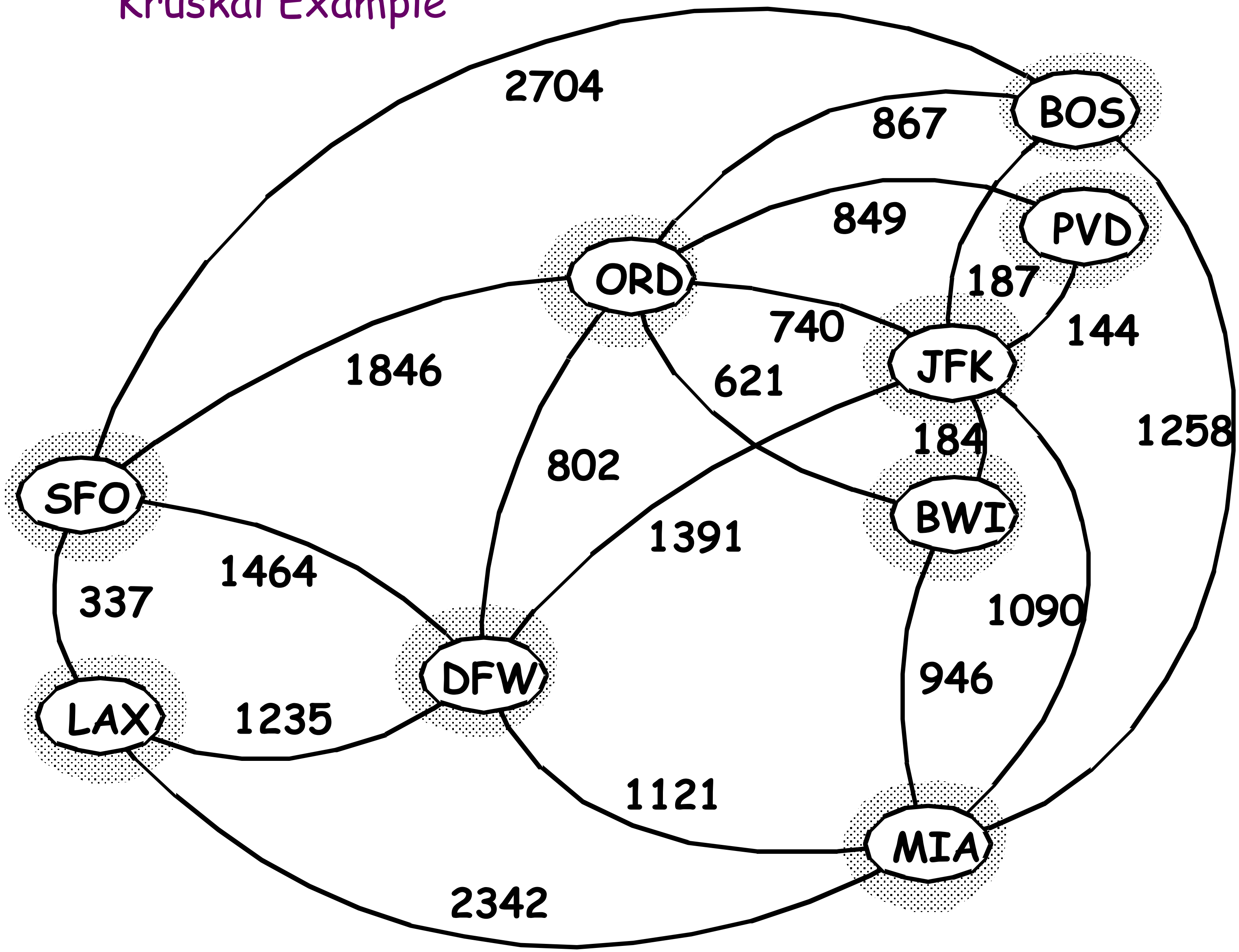
Kruskal's algorithm

```
T = empty spanning tree;
E = set of edges;
N = number of nodes in graph;
while T has fewer than N - 1 edges {
remove an edge (v, w) of lowest cost from E
if adding (v, w) to T would create a cycle
   then discard (v, w)
   else add (v, w) to T
}
```

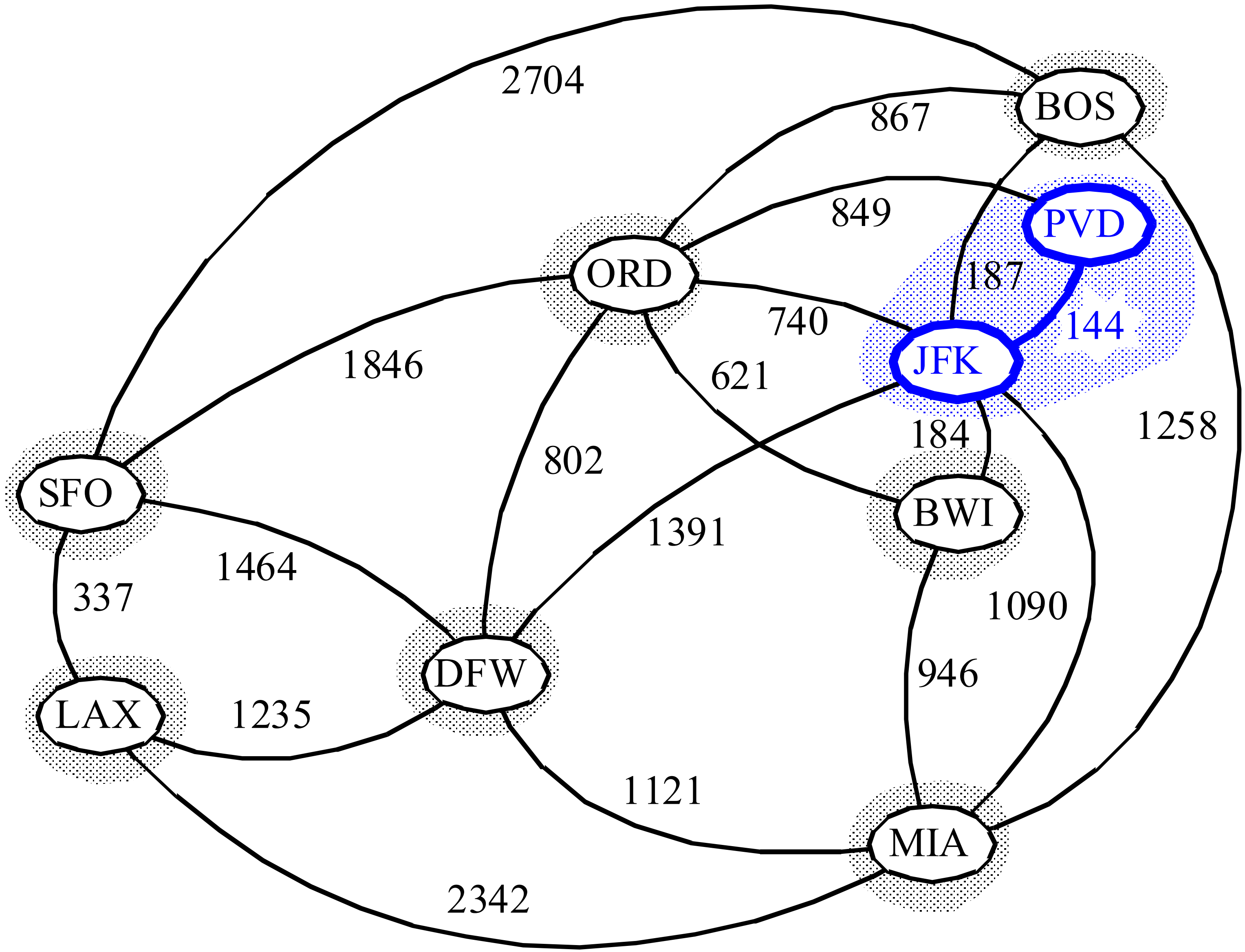
Finding an edge of lowest cost can be done just by sorting the edges

Testing for a cycle: Efficient testing for a cycle requires a additional algorithm (UNION-FIND) which we don't cover in this course. The main idea: If both nodes v, w are in the same component of T, then adding (v, w) to T would result in a cycle.

Kruskal Example

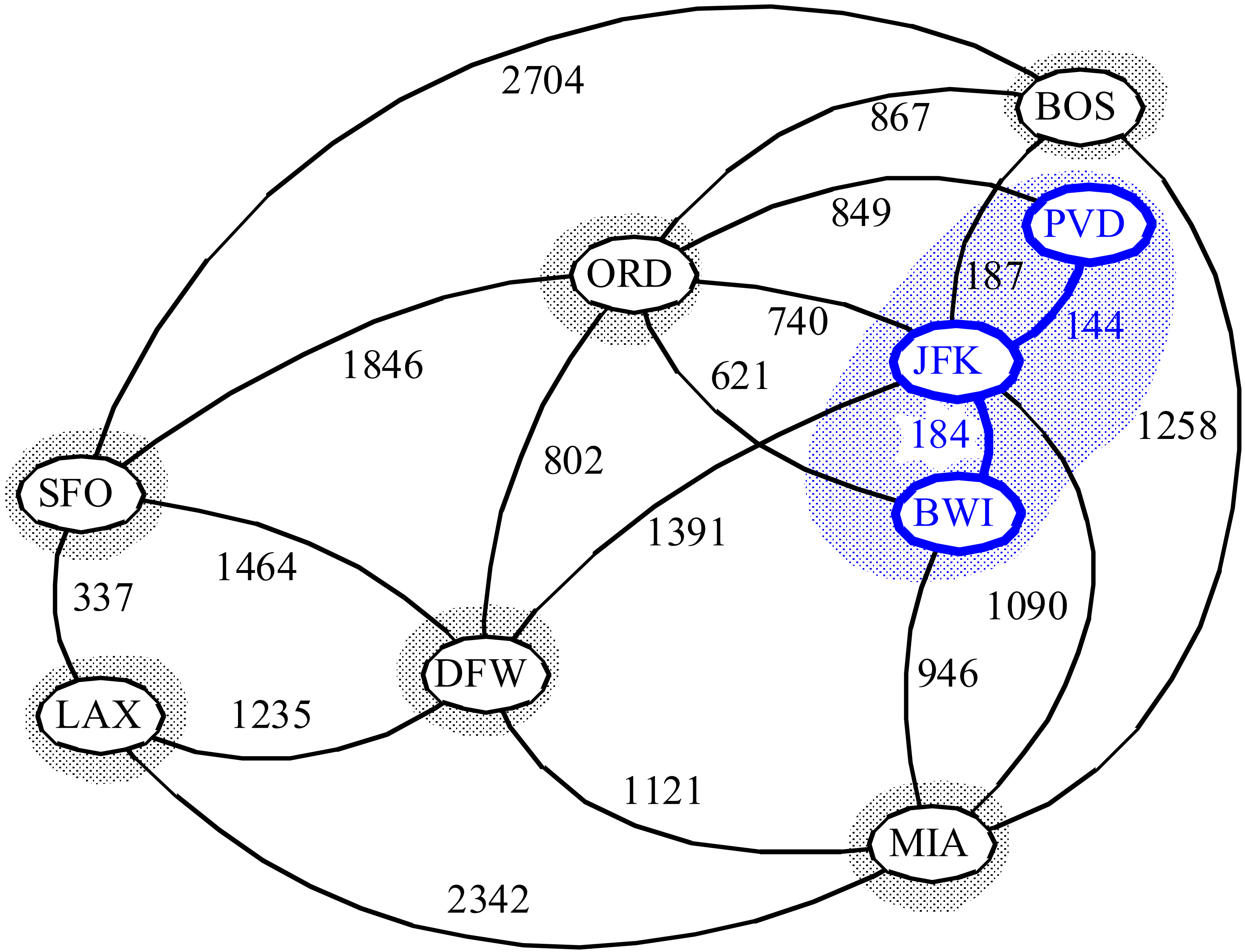


Example



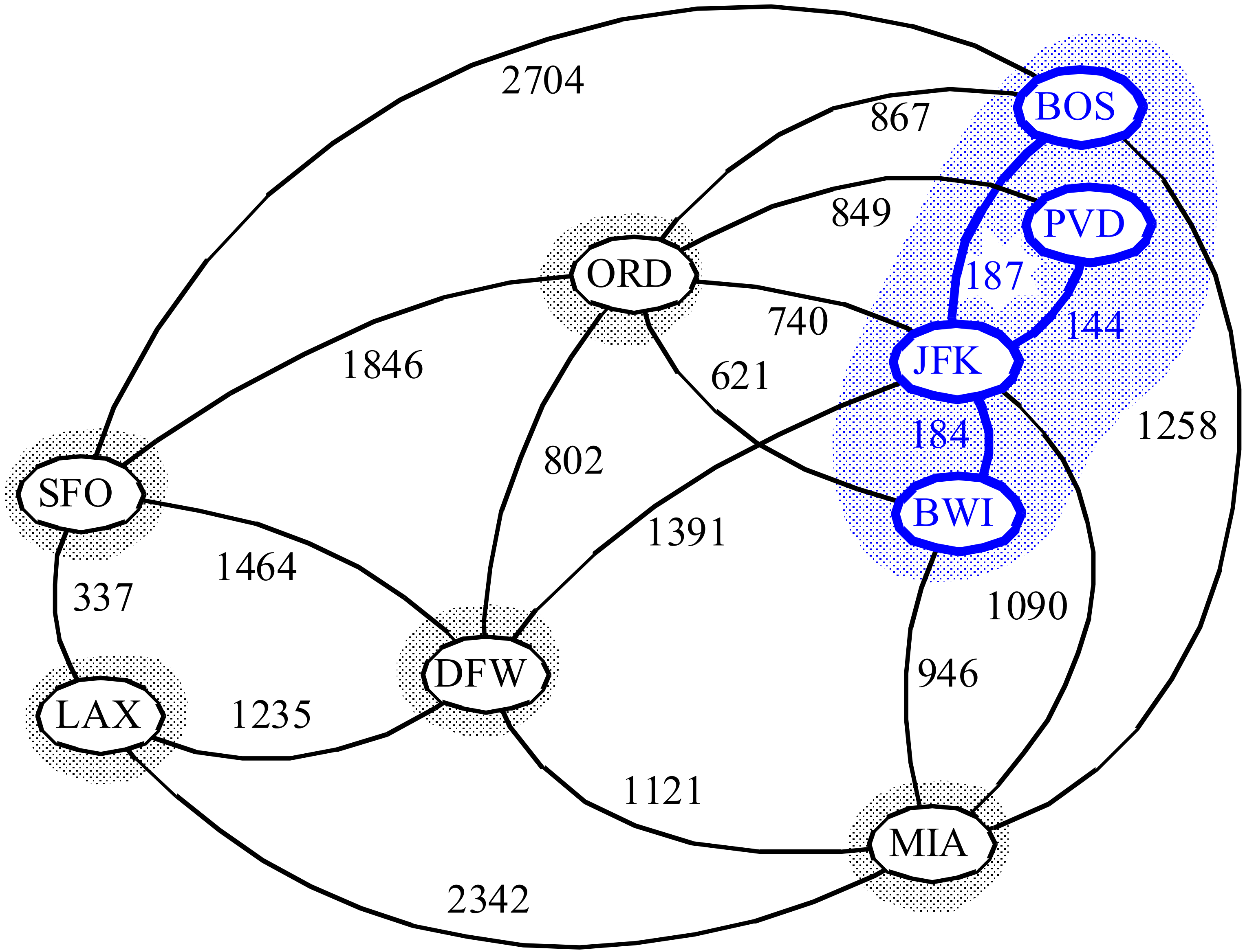
2704
BOS
867
849
PVD
ORD
187
740
144
1846
621
JFK
184
1258
SFO
802
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



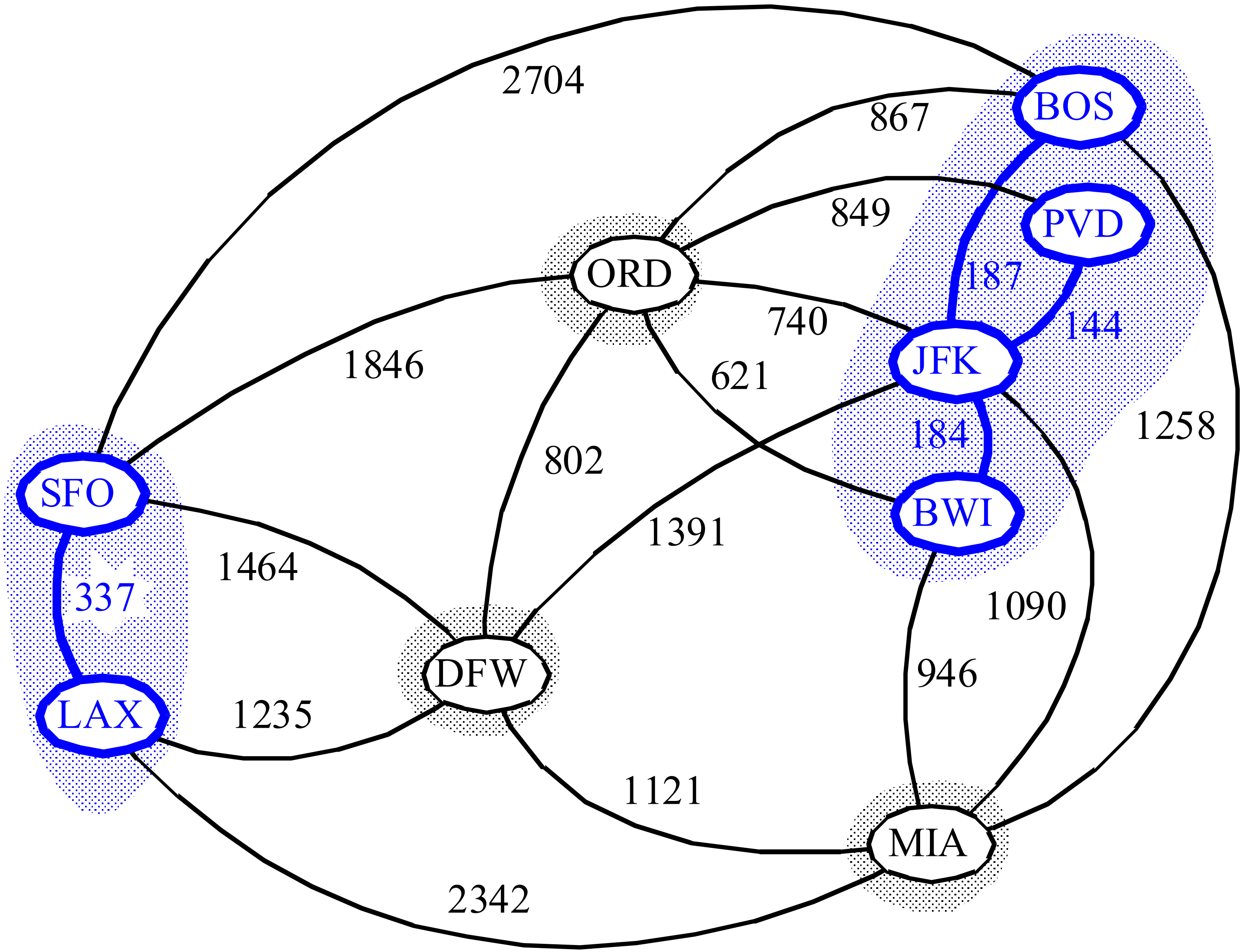
2704
BOS
867
849
PVD
ORD
187
740
144
1846
JFK
621
184
1258
SFO
802
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



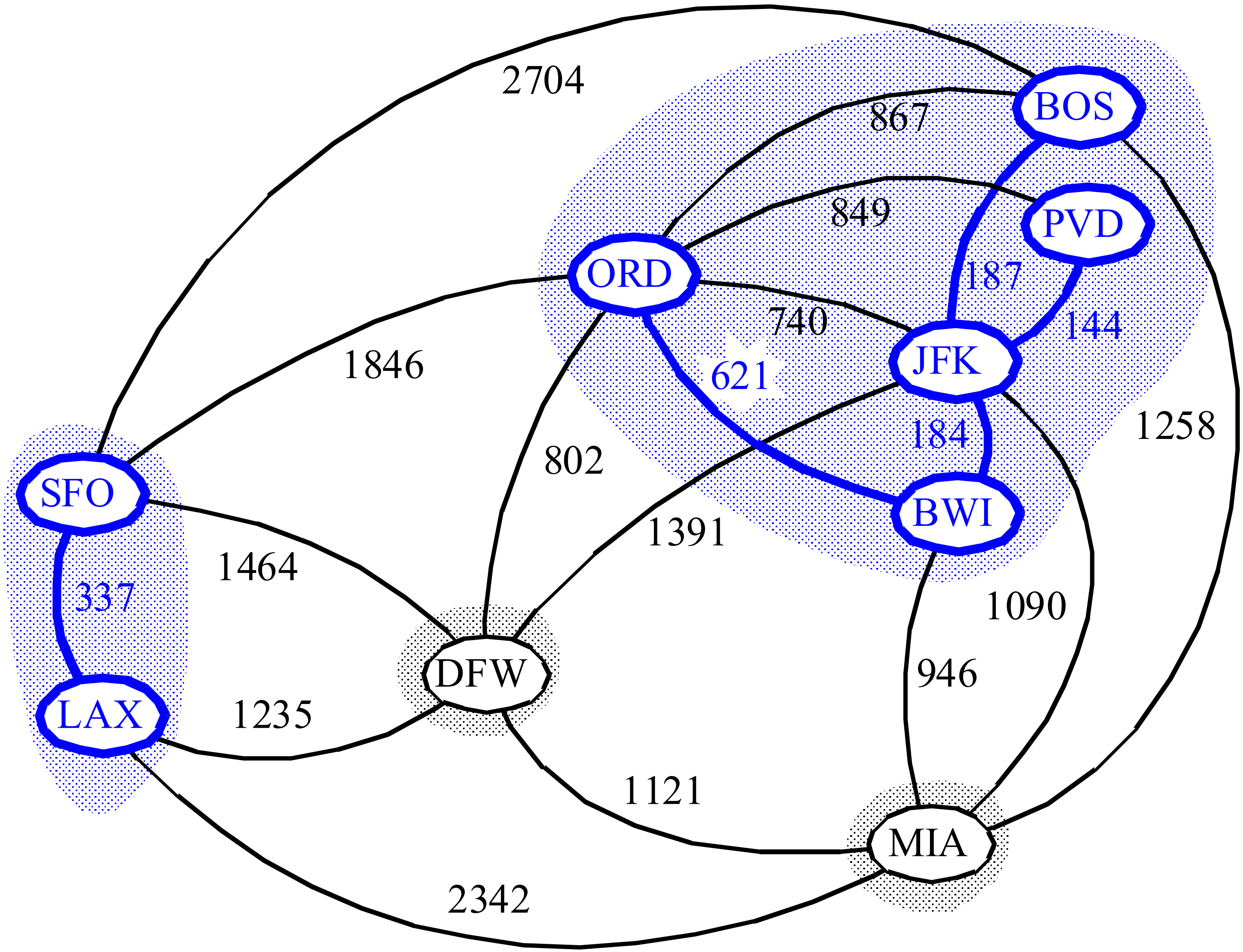
2704
BOS
867
PVD
849
ORD
187
740
144
JFK
1846
621
184
1258
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example

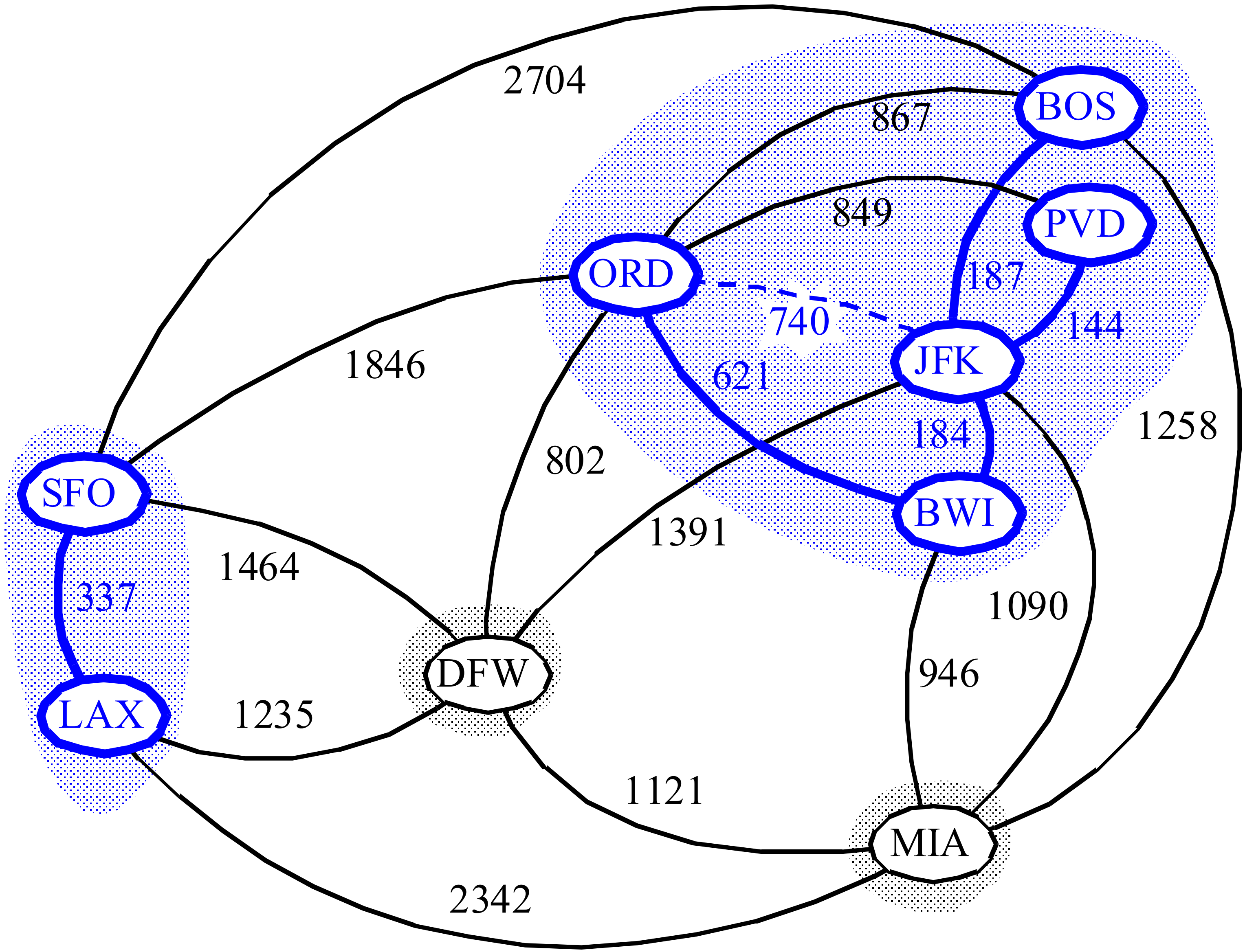


2704
BOS
867
849
PVD
ORD
187
740
144
1846
621
JFK
1258
SFO
802
184
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example

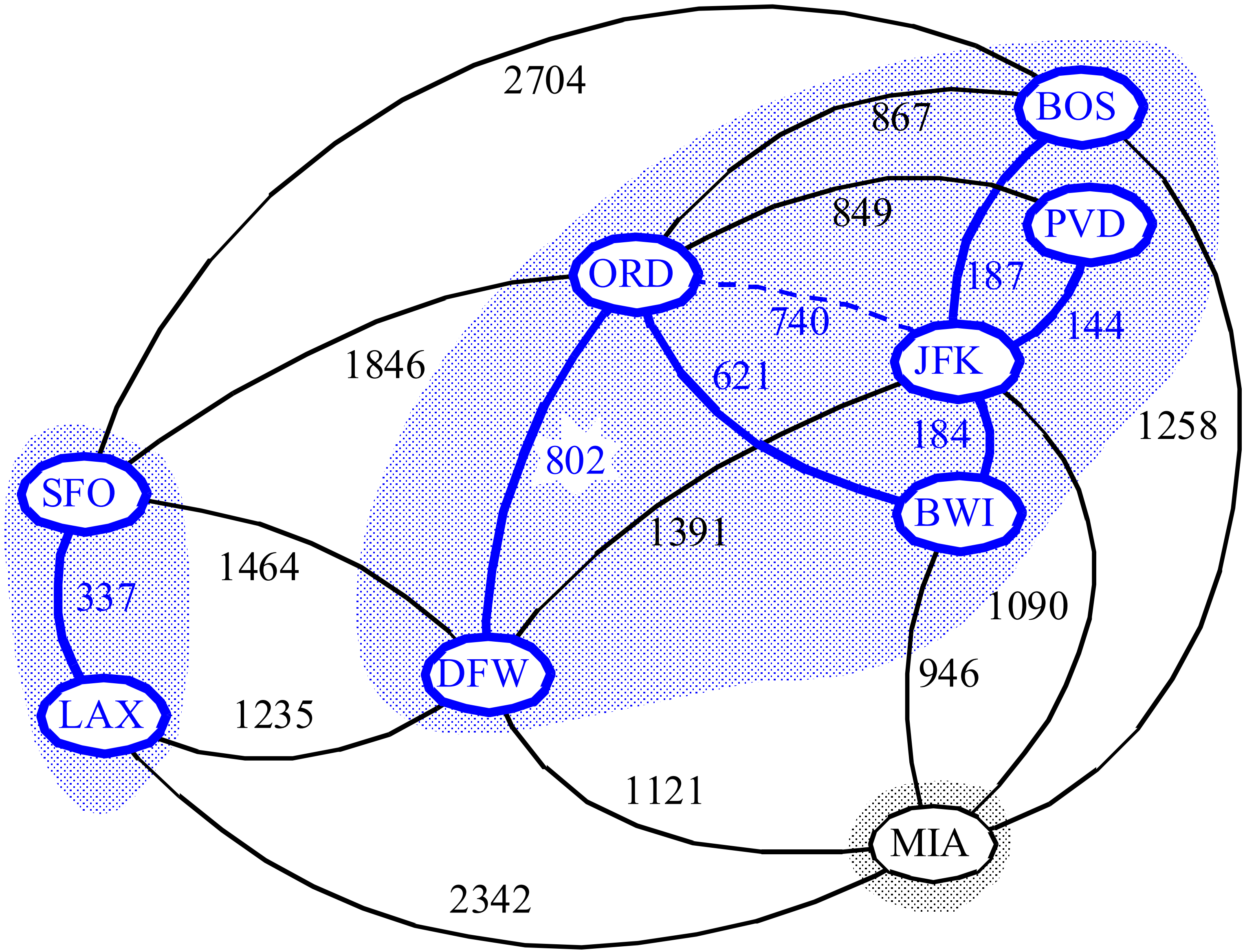

2704
BOS
867
849
PVD
ORD
187
740
144
1846
JFK
621
184
1258
SFO
802
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



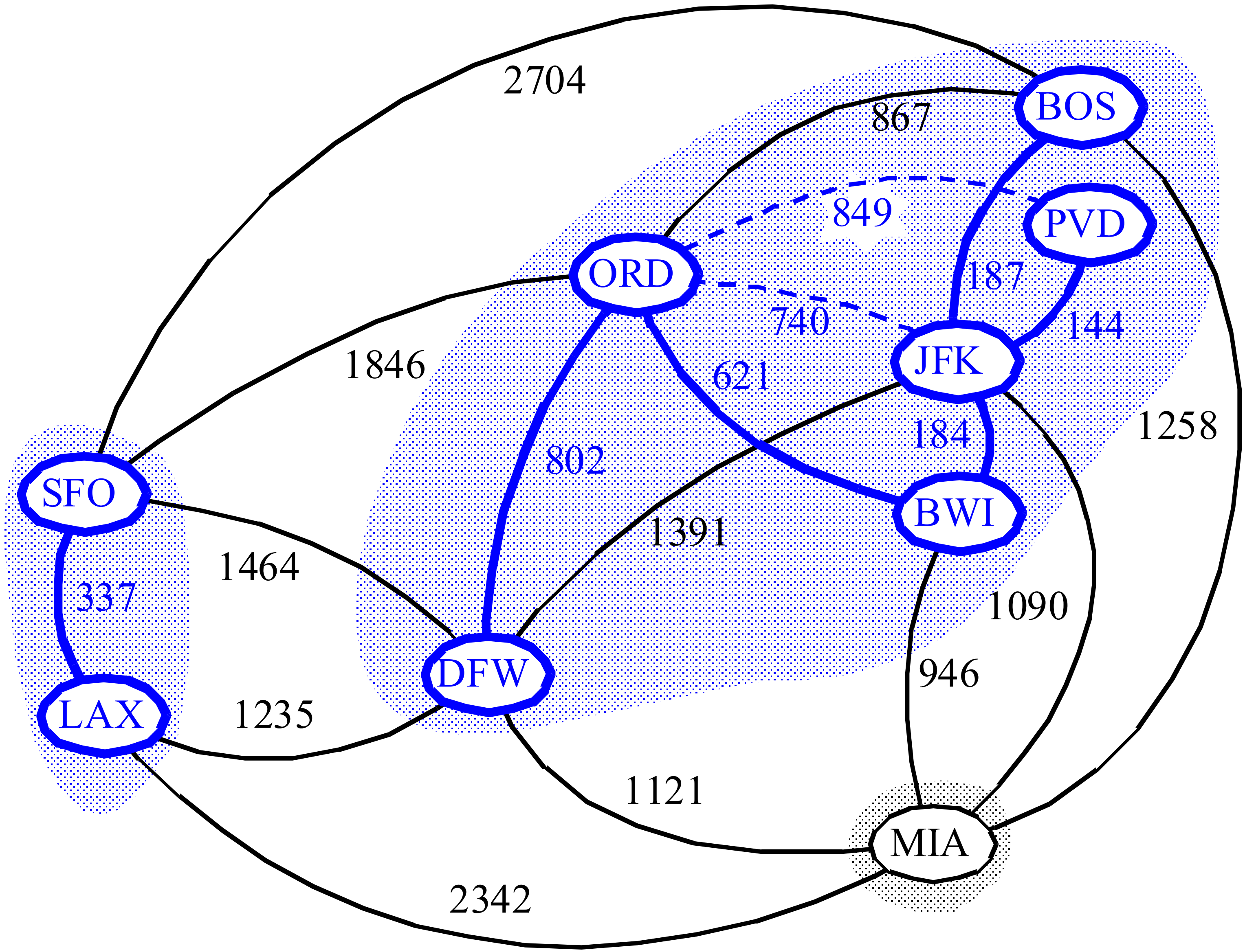
2704
867
BOS
849
PVD
ORD
187
740
144
JFK
1846
621
1258
184
802
SFO
BWI
1391
1464
1090
337
DFW
946
LAX
1235
1121
MIA
2342

Example



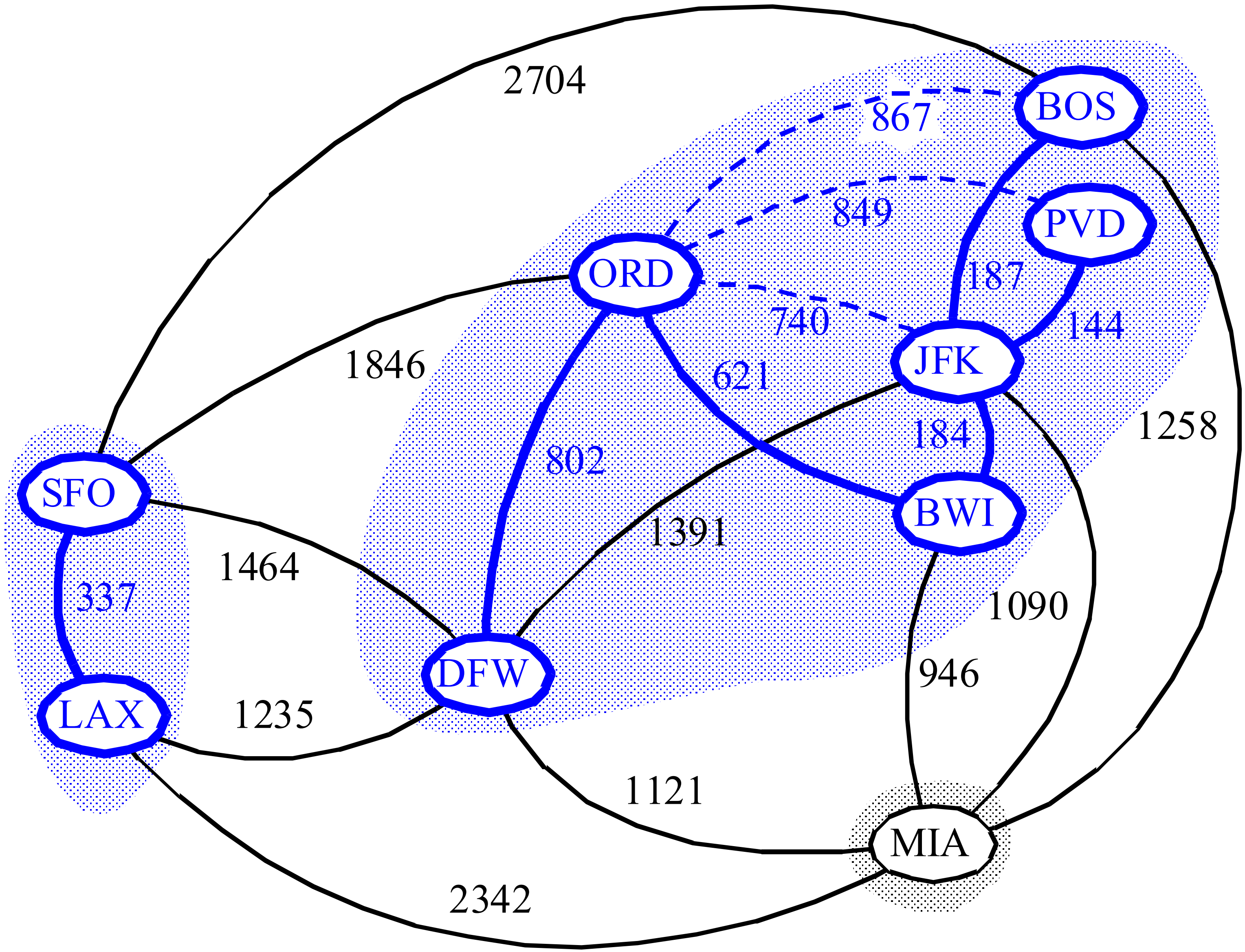
2704
867
849
BOS
PVD
ORD
187
740
144
JFK
1846
621
184
802
1258
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



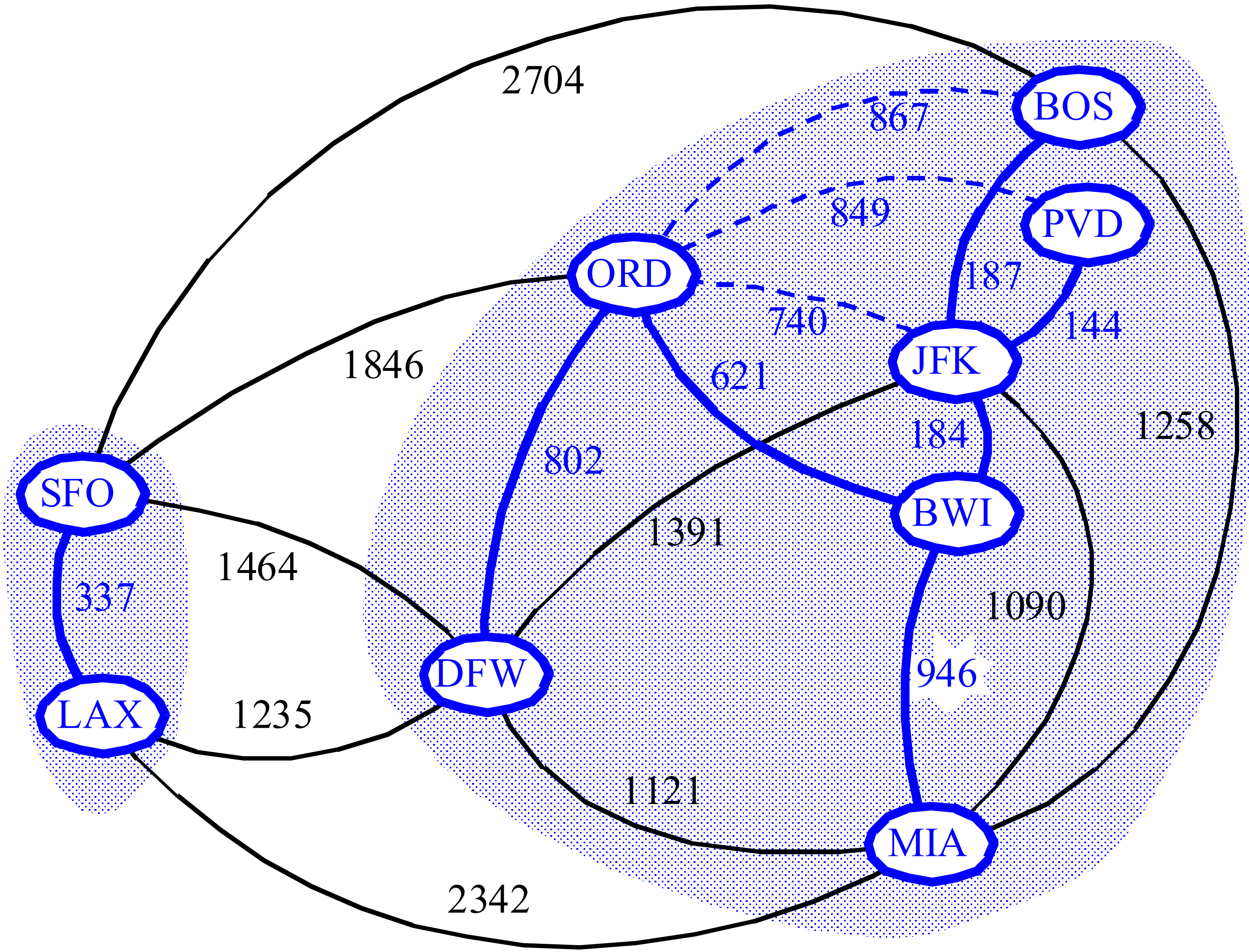
2704
BOS
867
849
PVD
ORD
187
740
144
1846
JFK
621
1258
184
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



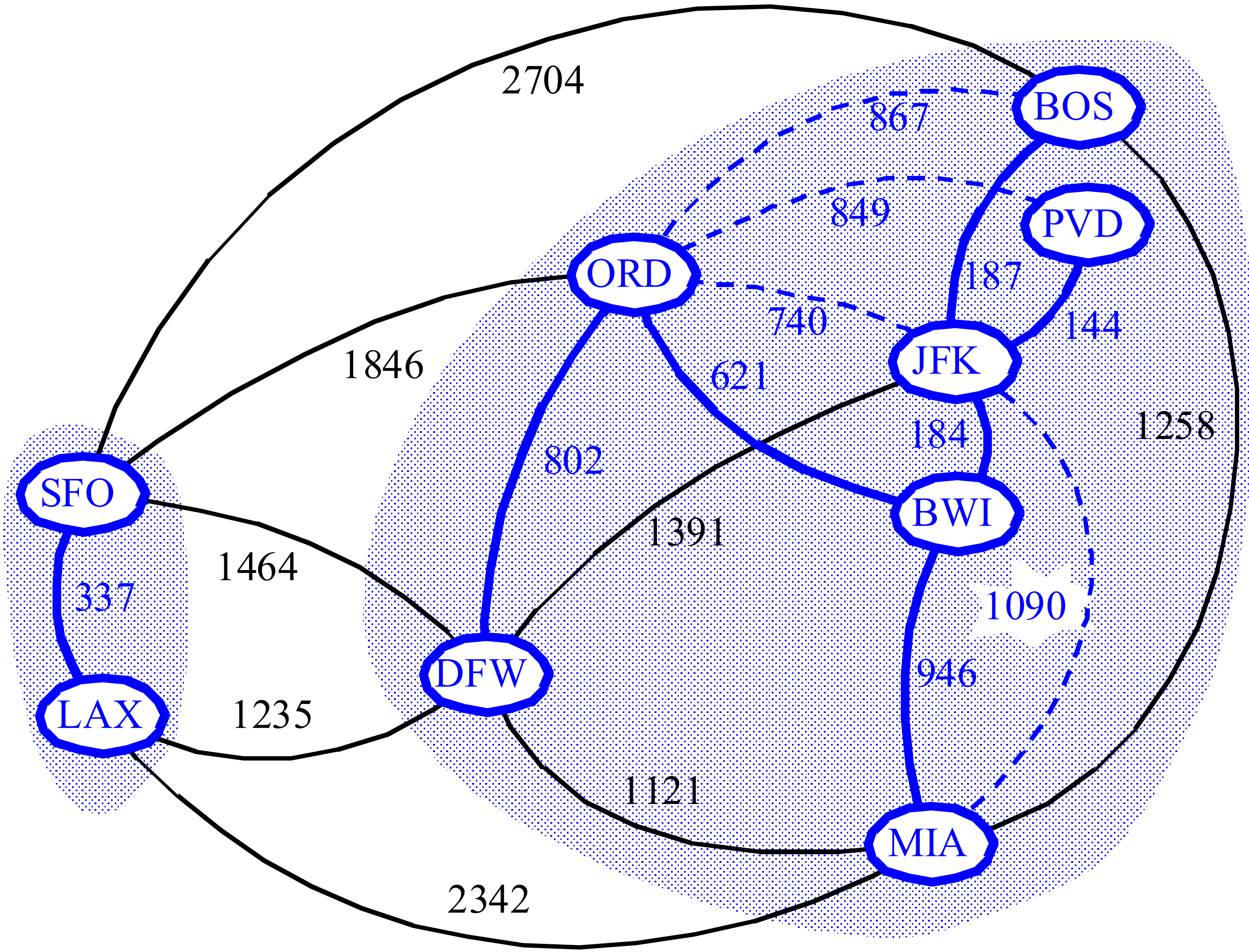
BOS
PVD
ORD
JFK
BWI
SFO
LAX
DFW
MIA
2704
867
849
187
144
740
1846
621
802
184
1258
1391
1464
337
1090
946
1235
1121
2342

Example



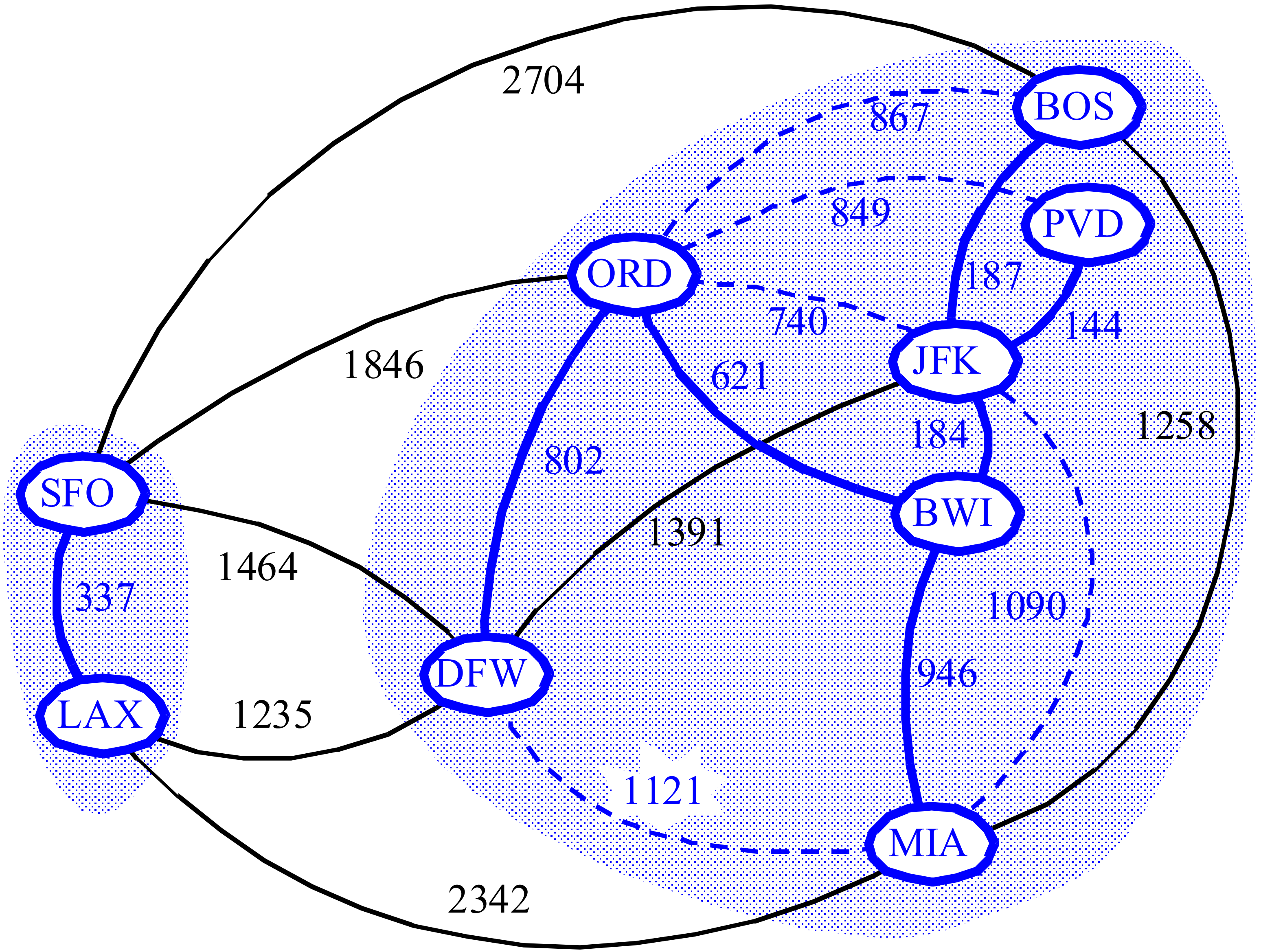
2704
BOS
867
849
PVD
ORD
187
740
144
1846
JFK
621
1258
184
802
SFO
BWI
1391
1464
337
1090
DFW
946
LAX
1235
1121
MIA
2342

Example



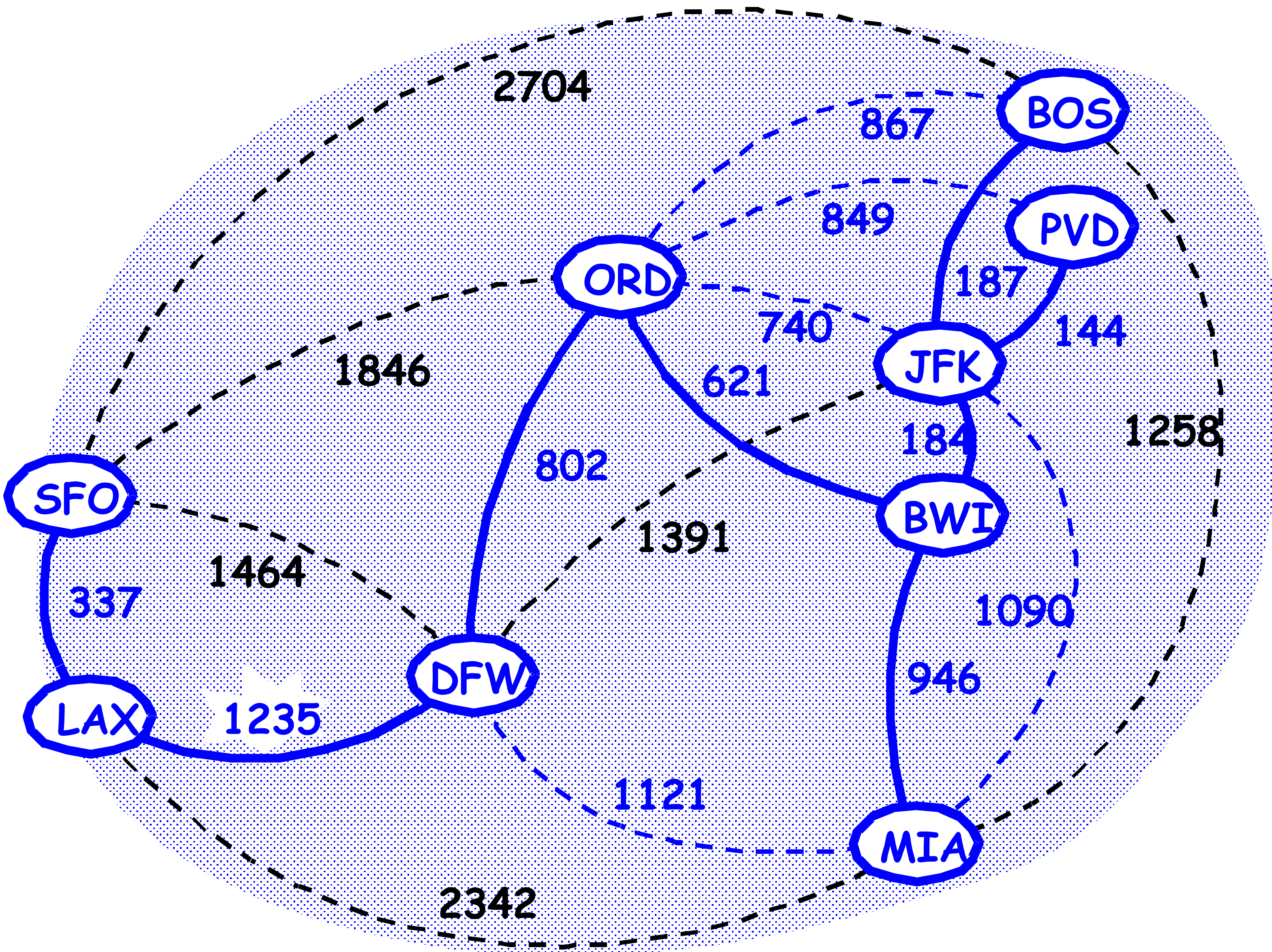
2704
BOS
867
849
PVD
ORD
187
740
144
1846
JFK
621
1258
184
802
SFO
BWI
1391
1464
337
1090
946
DFW
LAX
1235
1121
MIA
2342

Example



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

Example



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

Time Compexity

Let v be number of vertices and e the number of edges of a given graph.

Kruskal's algorithm: $O(e \log e)$

Prim's algorithm: $O(e \log v)$

Kruskal's algorithm is preferable on **sparse graphs**, i.e., where *e* is very small compared to the total number of possible edges: $C(v, 2) = v(v-1)/2$.

MST with Prim's and Kruskal algorithm