COMP9024: Data Structures and Algorithms

Week Twelve: Graphs (II)

Hui Wu

Session 1, 2015
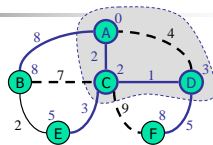http://www.cse.unsw.edu.au/~cs9024

1

Outline

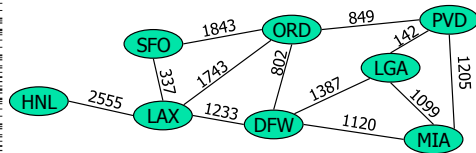- Shortest Paths
- Minimum Spanning Trees

2

Shortest Paths

3

Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports

4

Shortest Paths

Given a weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v.
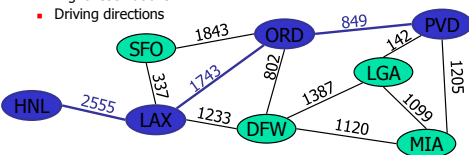- Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions

5
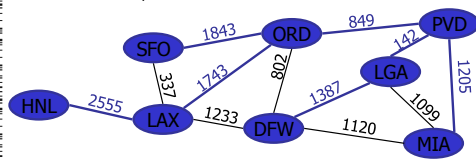
Shortest Path Properties

Property 1:
A subpath of a shortest path is itself a shortest path
Property 2:
There is a tree of shortest paths from a start vertex to all the other vertices
Example:
Tree of shortest paths from Providence

6

1

Dijkstra's Algorithm

- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative

- We grow a "**cloud**" of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $D(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label, $D(u)$
  - We update the labels of the vertices that are adjacent to u and not in the cloud

7

Edge Relaxation

- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
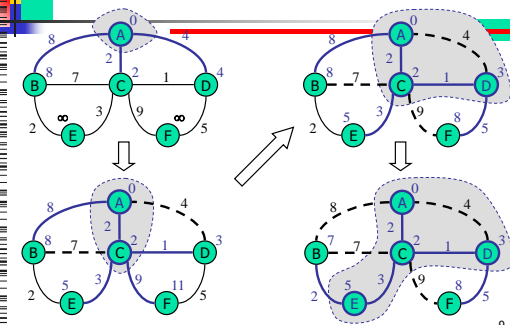 - z is not in the cloud
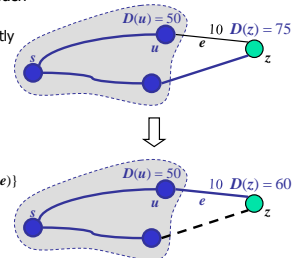
- The relaxation of edge e updates distance $D(z)$ as follows:
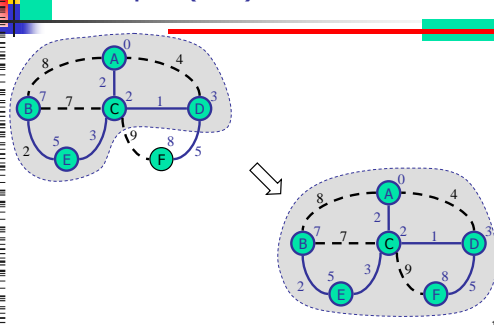
$$D(z) = \min\{D(z), D(u) + weight(e)\}$$

8

Example (1/2)

9

Example (2/2)

10

Dijkstra's Algorithm

Algorithm *DijkstraDistances(G, s)*
Input: A simple undirected weighted graph G with nonnegative edge weights, and a distinguished vertex s.
Output: A label D[u], for each vertex u, such that D[u] is the length of a shortest path from s to u in G.
{ for each $v \in G$ do
    if ($v = s$) do
        $D[v] = 0$;
    else
        $D[v] = +\infty$;
    Create a priority queue Q containing all the vertices of G using the D labels as keys;
    while *Q is not empty* do
    { $u = Q.removeMin()$;
        for each $z \in Q$ *such that z is adjacent to u* do
            if ($D[u] + w((u,z)) < D[z]$) // relax edge e
            { $D[z] = D[u] + w((u,z))$;
                Change to $D[z]$ the key of vertex z in Q;
            }
    }
   **return the label** $D[u]$ **of each vertex** u;
}

11

Analysis of Dijkstra's Algorithm

- Creating the priority queue Q takes $O(n \log n)$ time if using an adaptable priority queue, or $O(n)$ time by using bottom-up heap construction.
- At each iteration of the **while** loop, we spend $O(\log n)$ time to remove vertex u from Q and $O(\text{degree}(u) \log n)$ time to perform the relaxation procedure on the edges incident on u.
- The overall running time of the while loop is
  $O(\Sigma_u (1+\text{degree}(u)) \log n) = O((n + m) \log n)$
  (Recall that $\Sigma_u \text{degree}(u) = 2m$)
- The running time can also be expressed as $O(m \log n)$ since the graph is connected

12

2

Why Dijkstra's Algorithm Works

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn't find all shortest distances. Let v be the first wrong vertex the algorithm processed.
  - When the previous node, u, on the true shortest path was considered, its distance was correct.
  - But the edge (u,v) was **relaxed** at that time!
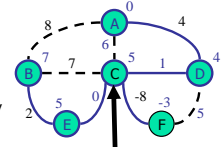 - Thus, so long as D(v)≥D(u), v's distance cannot be wrong. That is, there is no wrong vertex.
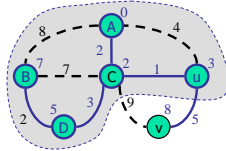
13

Why It Doesn't Work for Negative-Weight Edges

◆ Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.

- If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.

C's true distance is 1, but it is already in the cloud with D(C)=5!

14

Bellman-Ford Algorithm (not in book)

- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.
- Running time: O(nm).
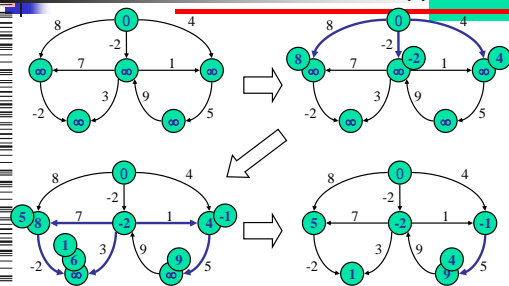- Can be extended to detect a negative-weight cycle if it exists
  - How?

```
Algorithm BellmanFord(G, s)
{  for each v ∈ G do
      if ( v = s )
         D[v] = 0;
      else
         D[v] = +∞;
   for ( i = 1;  i ≤ n-1;  i ++ )
      for each  e ∈ G.edges()
         // relax edge e
         { u = G.origin(e);
            z = G.opposite(u,e);
            r = D[u] + weight(e);
            if ( r < D[z] )
               D[z] = r;
         }
}
```

15

Bellman-Ford Example
Nodes are labeled with their D(v) values

16

DAG-based Algorithm (not in book)

- Works even with negative-weight edges
- Uses topological order
- Doesn't use any fancy data structures
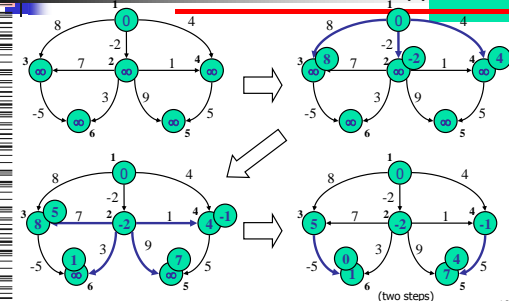- Is much faster than Dijkstra's algorithm
- Running time: O(n+m).

```
Algorithm DagDistances(G, s)
{  for all  v ∈ G.vertices()
      if ( v = s )
         D[v] = 0;
      else
         D[v] = +∞;
   Perform a topological sort of the vertices;
   for ( u = 1;  u ≤ n;  u ++ )
      // in topological order
      for each  e ∈ G.outEdges(u) do
         // relax edge e
         { z = G.opposite(u,e);
            r = D[u] + weight(e);
            if ( r < D[z] )
               D[z] = r;
         }
}
```
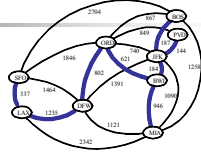
17

DAG Example
Nodes are labeled with their D(v) values

(two steps)

18

3

Minimum Spanning Trees

Minimum Spanning Trees

- Spanning subgraph
  - Subgraph of a graph G containing all the vertices of G
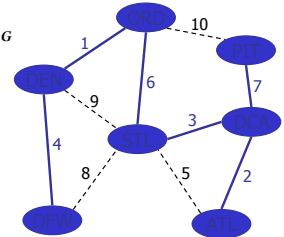- Spanning tree
  - Spanning subgraph that is itself a (free) tree
- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight
- Applications
 - Communications networks
  - Transportation networks

Cycle Property

Cycle Property:
- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

Proof:
- By contradiction
- If $weight(f) > weight(e)$ we can get a spanning tree of smaller weight by replacing e with f

Replacing f with e yields a better spanning tree

---
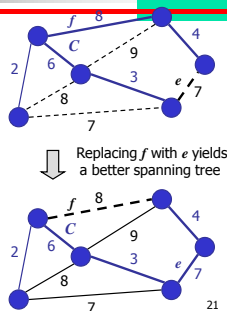
Partition Property

Partition Property:
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:
- Let T be an MST of G
- If T does not contain e, consider the cycle C formed by e with T and let f be an edge of C across the partition
- By the cycle property, $weight(f) \leq weight(e)$
- Thus, $weight(f) = weight(e)$
- We obtain another MST by replacing f with e

Replacing f with e yields another MST
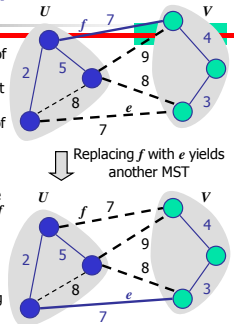
Kruskal's Algorithm

- A priority queue stores the edges outside the cloud
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - We are left with one cloud that encompasses the MST
  - A tree T which is our MST

```
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 = T.removeMin();
    Let u, v be the endpoints of e;
    if ( Cloud(v) ≠ Cloud(u) )
    { Add edge e to T;
      Merge Cloud(v) and Cloud(u);
    }
  }
  return T;
}
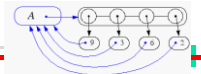```

---

# Data Structure for Kruskal Algorithm

- The algorithm maintains a forest of trees
- An edge is accepted it if connects distinct trees
- We need a data structure that maintains a **partition**, i.e., a collection of disjoint sets, with the operations:
  - **find**(u): return the set storing u
  - **union**(u,v): replace the sets storing u and v with their union

## Representation of a Partition



- Each set is stored in a sequence
- Each element has a reference back to the set
  - operation find(u) takes O(1) time, and returns the set of which u is a member.
  - in operation union(u,v), we move the elements of the smaller set to the sequence of the larger set and update their references
  - the time for operation union(u,v) is min($n_u$,$n_v$), where $n_u$ and $n_v$ are the sizes of the sets storing u and v
- Whenever an element is processed, it goes into a set of size at least double, hence each element is processed at most log n times

25

## Partition-Based Implementation

A partition-based version of Kruskal's Algorithm performs cloud merges as unions and tests as finds.
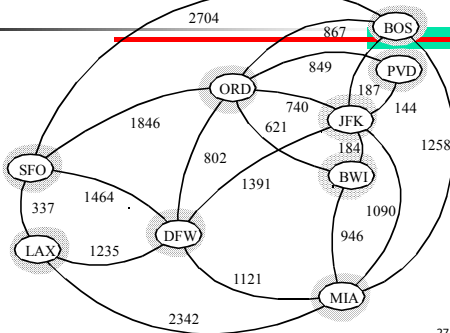
```
Algorithm Kruskal(G):
  Input: A weighted graph G.
  Output: An MST T for G.
{ Let P be a partition of the vertices of G, where each vertex forms a separate set;
  Let Q be a priority queue storing the edges of G, sorted by their weights;
  Let T be an initially-empty tree;
  while Q is not empty do
    { (u,v) = Q.removeMinElement();
      if (P.find(u) != P.find(v))
        { Add (u,v) to T;
          P.union(u,v);
        }
    }
  return T;
}
```
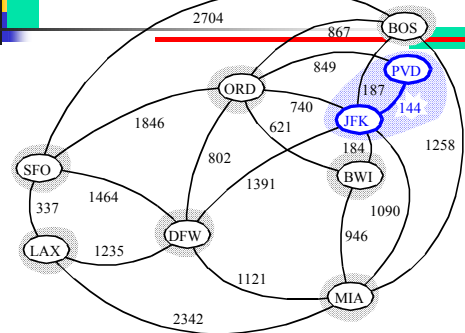
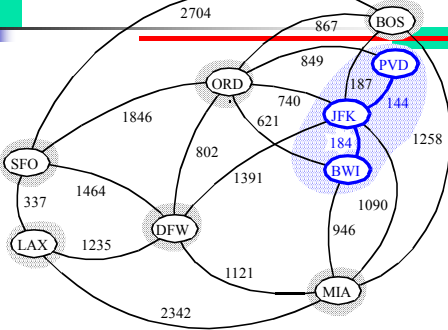Running time: O((m+n) log n)

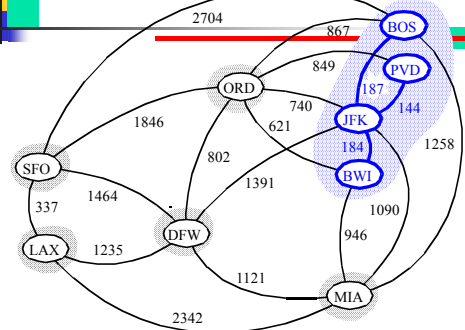26

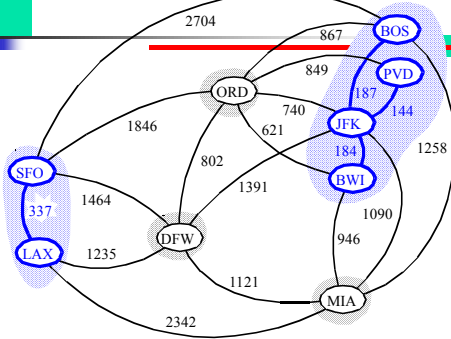## Kruskal Example (1/14)



27

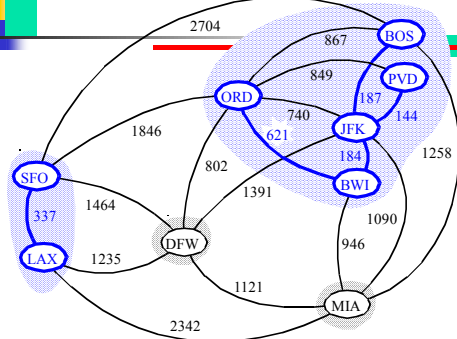## Example (2/14)



28

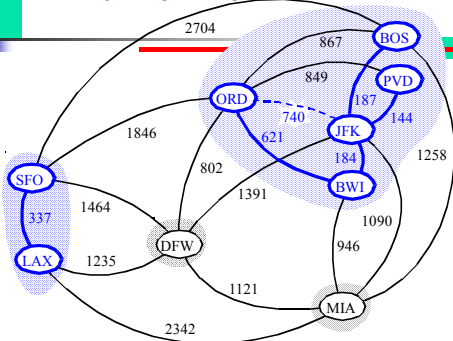## Example (3/14)



29

## Example (4/14)



30

Example (5/14)

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

31

Example (6/14)

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

32

Example (7/14)

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

33

Example (8/14)

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

34

Example (9/14)

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

35

Example (10/14)

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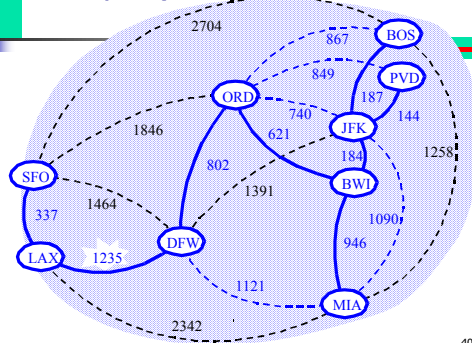802  184  1258
SFO  1391  BWI
337  1464  1090
DFW  946
LAX  1235
1121  MIA
2342

36

6

## Prim-Jarnik's Algorithm (1/2)

- 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 *v*
- We store with each vertex *u* a label $D(u)$ = the smallest weight of an edge connecting *u* 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*
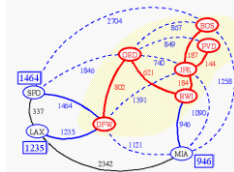


41

## Prim-Jarnik's Algorithm (2/2)

**Algorithm** *PrimJarnikMST(G)*
```
{
    Pick any vertex v of G;
    D[v] = 0;
    for each u ∈ G with u ≠ v do
        D[u] = +∞;
    T = Ø ;
    Create a priority queue Q with an entry
        ((u, null), D[u]) for each vertex u, where
        (u, null) is the element and D[u] is the key;
    while Q is not Empty do
    { (u, e) = Q.removeMin();
        add vertex u and edge e to T;
        for each vertex z in Q such that z is adjacent to u do
            if (w((u, z)) < D[z])
            { D[z] = w((u, z));
                Change to (z, (u,z)) the element of vertex z in Q;
                Change to D[z] the key of vertex z in Q;
            }
    }
    return T;
}
```
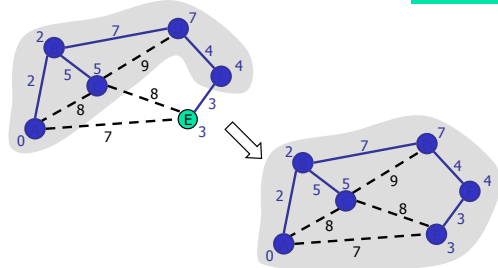
42

## Example (1/2)
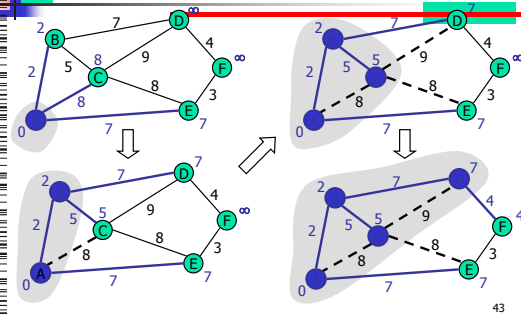
## Example (2/2)

## Analysis

- Graph operations
  - Method incidentEdges is called once for each vertex
- Label operations
  - We set/get the distance, parent and locator labels of vertex $z$ $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex $w$ in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Prim-Jarnik's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$
- The running time is $O(m \log n)$ since the graph is connected

## Baruvka's Algorithm (not in book)

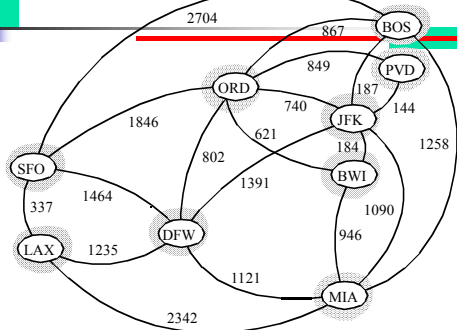- Like Kruskal's Algorithm, Baruvka's algorithm grows many "clouds" at once.

```
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;
}
```

- Each iteration of the while-loop halves the number of connected components in T.
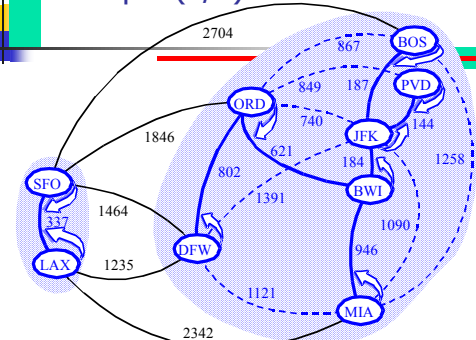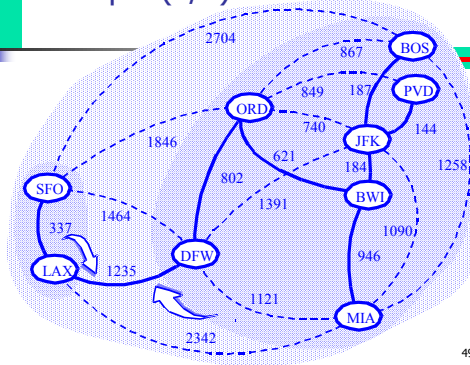 - The running time is O(m log n).

Baruvka Example (1/3)

Example (2/3)

Example (3/3)

2704
867
849 187
ORD 740
1846
621
802
SFO 1464 1391
337
LAX 1235 DFW
1121
2342

BOS
PVD
JFK 144
184 1258
BWI
1090
946
MIA

References

1. Chapter 13, Data Structures and Algorithms by Goodrich and Tamassia.