Minimum Spanning Tree

- Terminology and Properties
- Prim’s Algorithm
- Union-Find
- Kruskal’s Algorithm
- Baruvka’s Algorithm
- Traveling Salesperson Problem

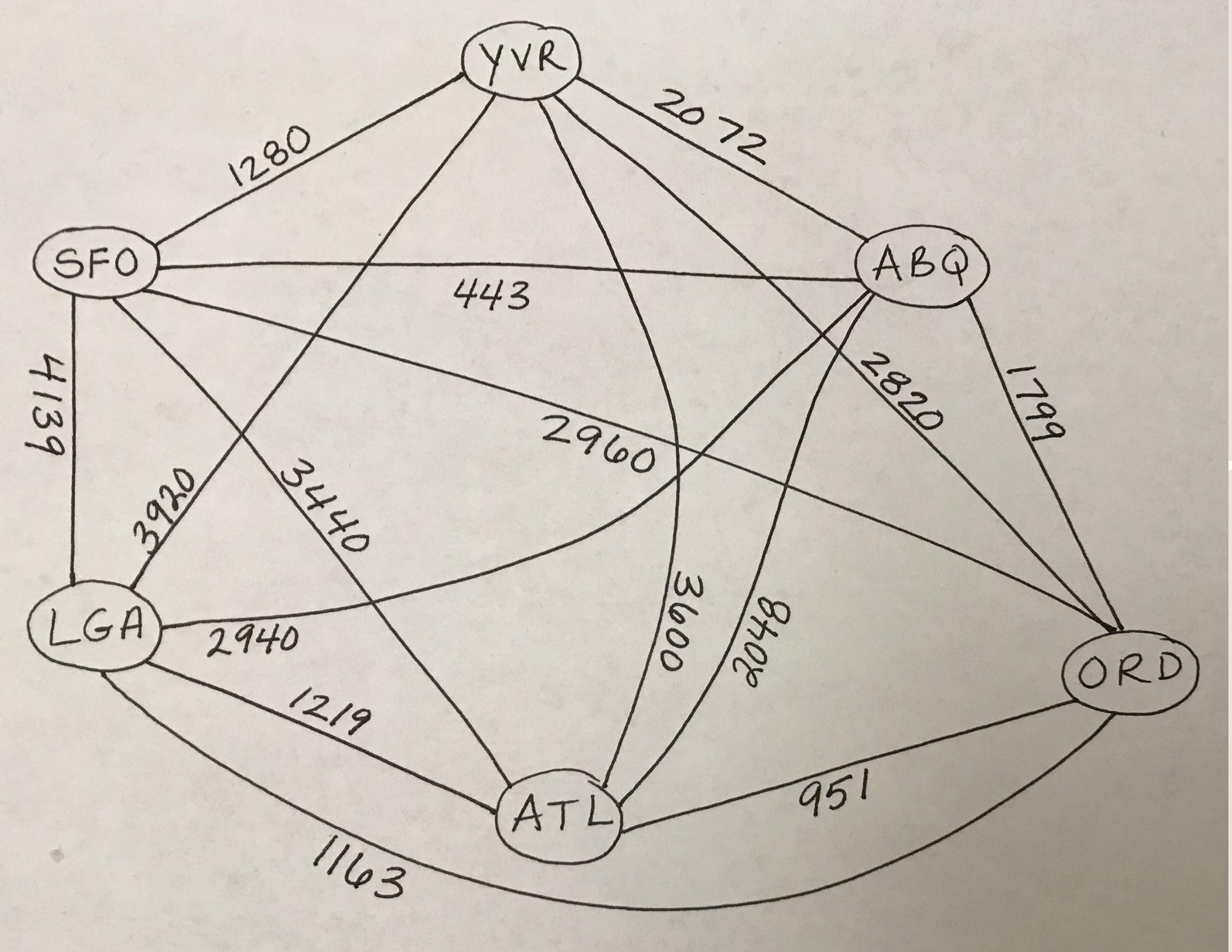
Imagine...

You are employed at a brand new airline, and your first task is to determine which flights the airline should provide according to the following guidelines:

- There needs to be a way to get between each pair of cities/airports in the following list: San Francisco, CA (SFO); Vancouver, BC (YVR); Albuquerque, NM (ABQ); Chicago, IL (ORD); Atlanta, GA (ATL); New York, NY (LGA)
- The route between two cities does not have to be a direct flight.
- The total number of flights should be minimized.
- The total distance covered by all flights should be minimized.
- Assume that if a flight exists, it goes both ways (e.g. SFO→ YVR and YVR→ SFO)

How would you go about solving this problem?

	SFO	YVR	ABQ	ORD	ATL	LGA
SFO	0	1280	443	2960	3440	4139
YVR	1280	0	2072	2820	3600	3920
ABQ	443	2072	0	1799	2048	2940
ORD	2960	2820	1799	0	951	1163
ATL	3440	3600	2048	951	0	1219
LGA	4139	3920	2940	1163	1219	0



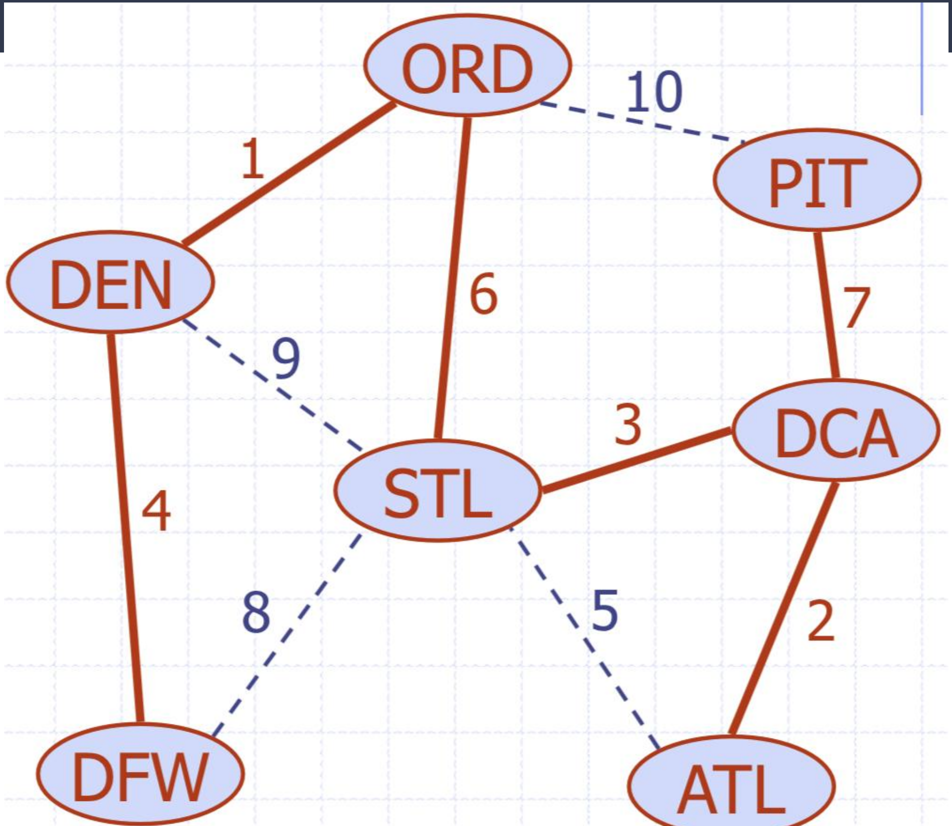
How does this problem differ from a shortest path problem?

In graph terms, how would you describe what we are looking for?

What we really want is a minimum spanning tree (MST)...

A minimum spanning tree of a graph G...

- ...is a spanning subgraph (i.e. it contains all the vertices of G)
- ...is a tree (i.e. it has no cycles)
- ...is minimal weight-wise (i.e. the total weight of all the edges it uses is minimized so that no other spanning tree has a smaller total weight)
- ...does NOT guarantee the shortest path between two vertices!



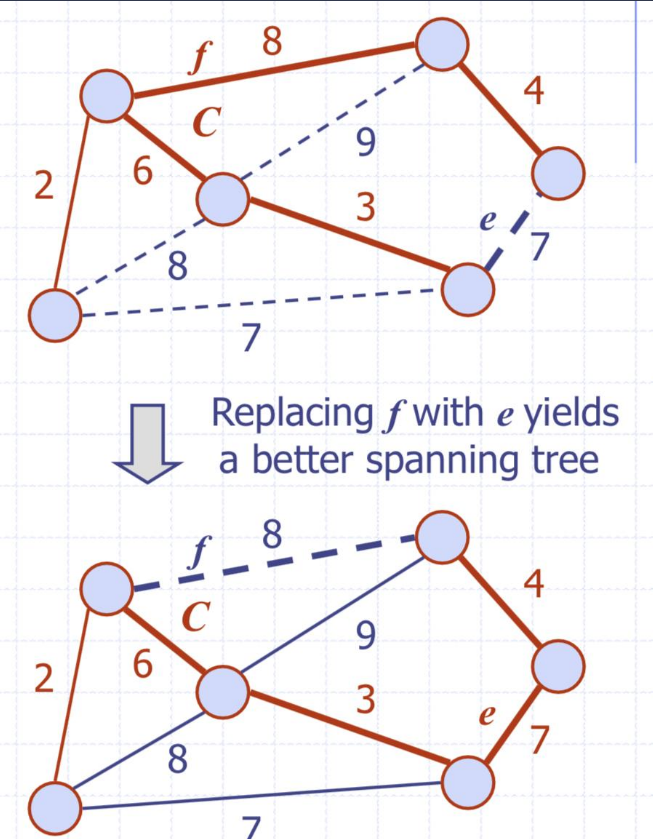
Picture from [1]

Property 1: The 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**
- Let **C** be the cycle formed by adding **e** to **T**
- Claim: For every edge **f** of **C**:

  weight(f) <= weight(e)

Proof by Contradiction: ???



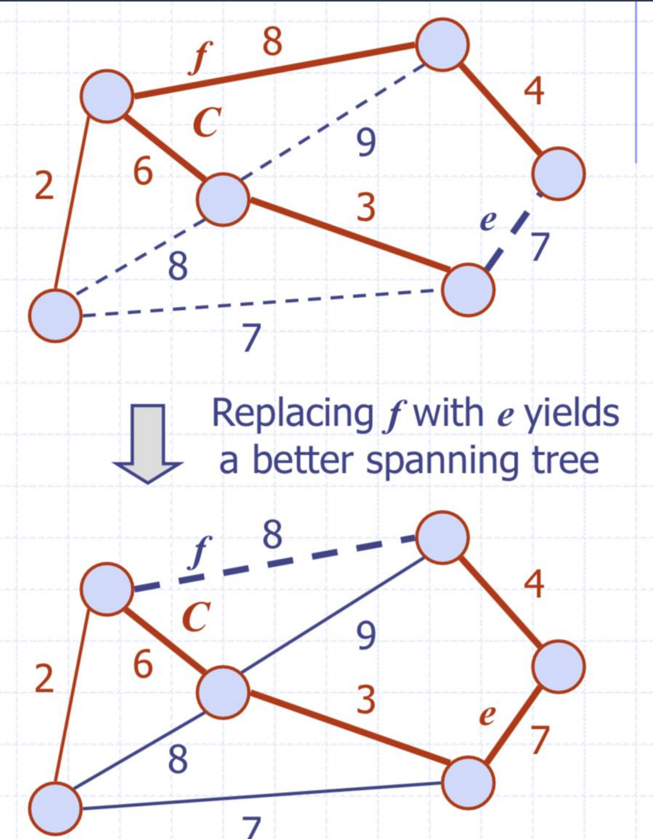
Picture from [1]

Property 1: The 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**
- Let **C** be the cycle formed by adding **e** to **T**
- Claim: For every edge **f** of **C**:

  weight(f) <= weight(e)

Proof by Contradiction: Given all the properties above, assume that for some edge **f** in **C**, ***weight(f) > weight(e).*** Then replacing **f** with **e** will produce a spanning tree **T'** such that the total weight of **T'** is smaller than the total weight of **T**. But that contradicts the definition of **T** as an MST of **G.**

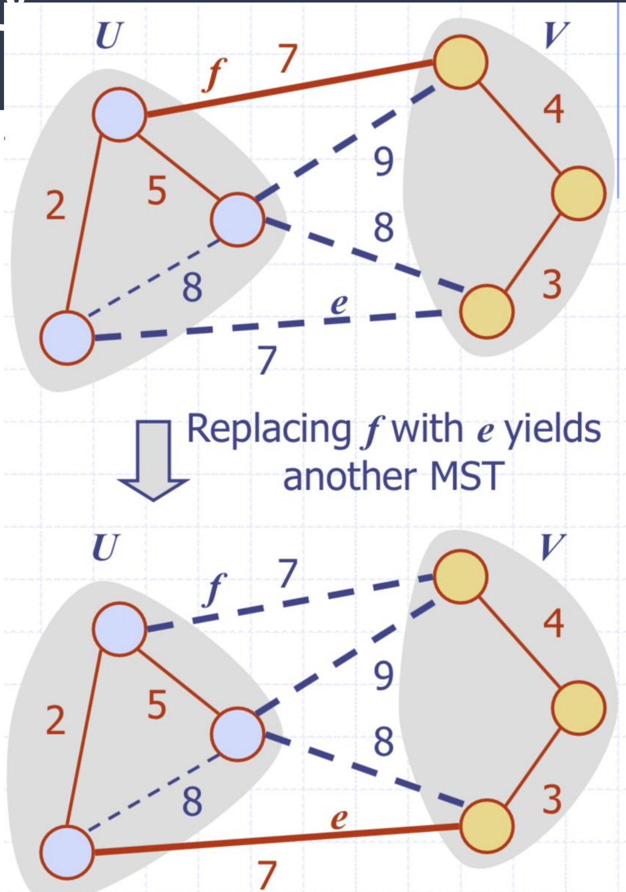


Picture from [1]

Property 1: The Partition Property

- Consider a partition of a graph **G** into subsets **U** and **V**
- Let **e** be an edge of minimum weight across the partition (i.e. **e** has one endpoint in **U** and one endpoint in **V**)
- Claim: There is a minimum spanning tree of **G** that includes edge **e**

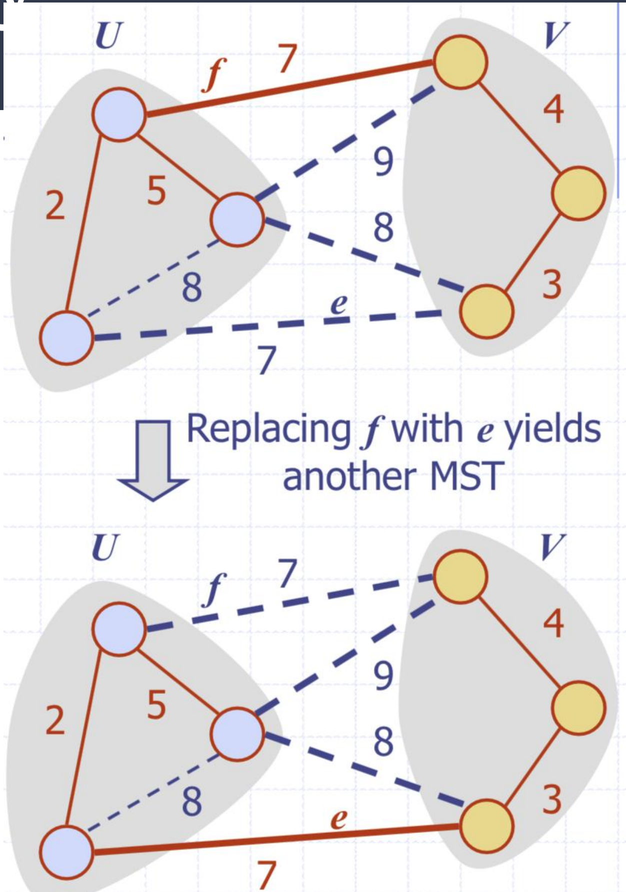
Proof: ???



Picture from [1]

Property 1: The Partition Property

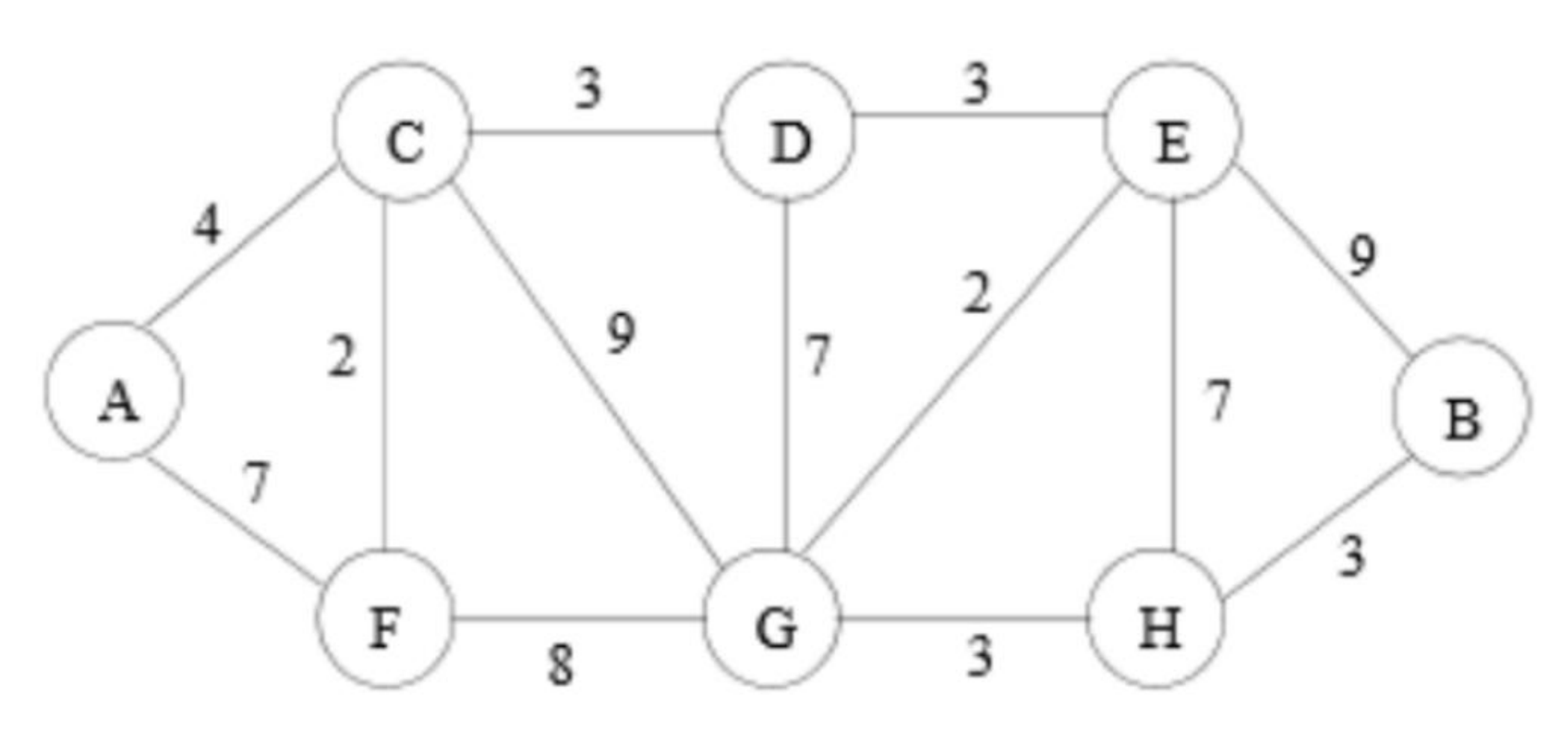
- Consider a partition of a graph **G** into subsets **U** and **V**
- Let **e** be an edge of minimum weight across the partition (i.e. **e** has one endpoint in **U** and one endpoint in **V**)
- Claim: There is a minimum spanning tree of **G** that includes edge **e**

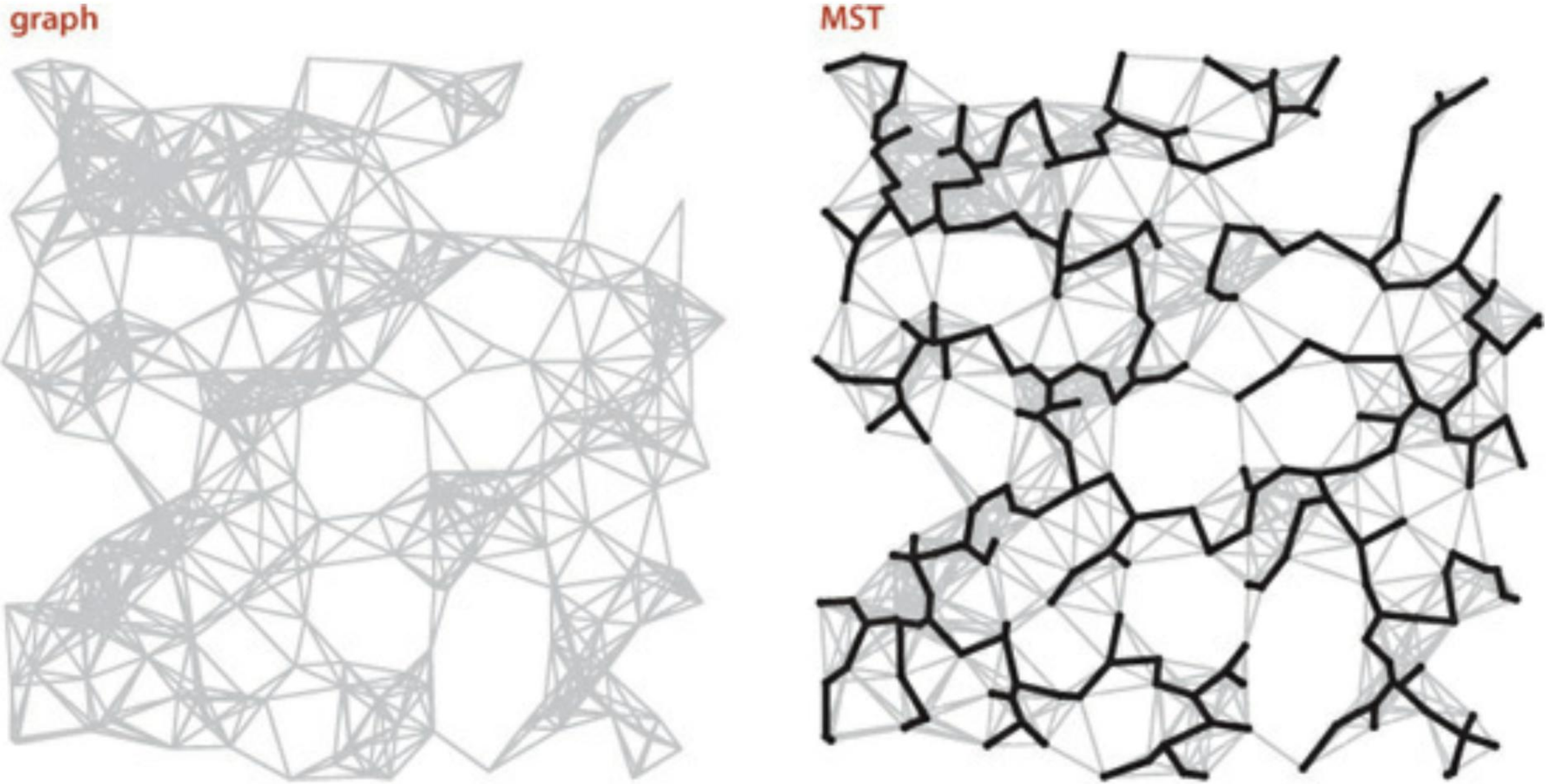


Picture from [1]

Proof: Let **T** be an MST of **G** and let the definitions above be true. 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)<=weight(e)***. Thus, ***weight(f) = weight(e)***, and we obtain another MST by replacing **f** with **e**.

So how would you find an MST of this graph?



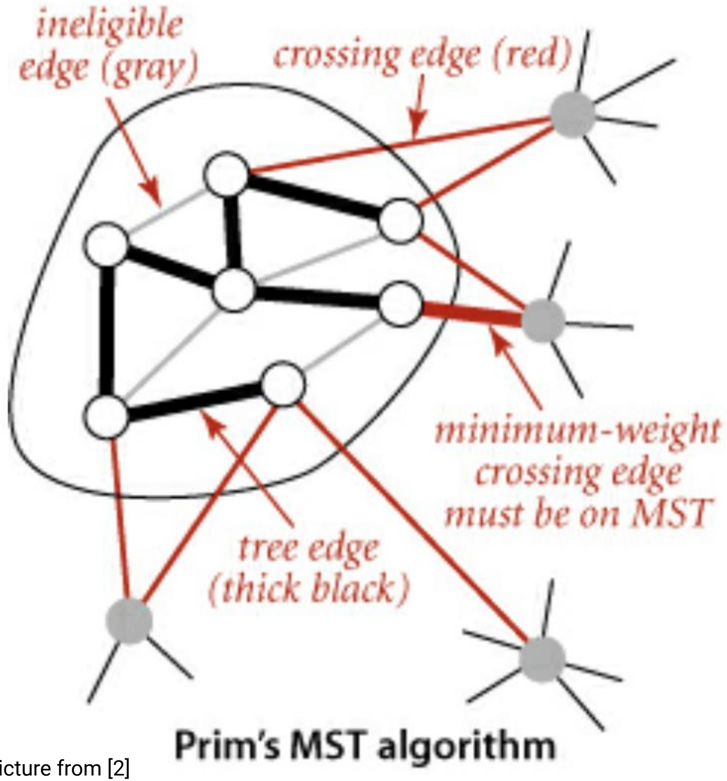


A 250-vertex Euclidean graph (with 1,273 edges) and its MST

Picture from [2]

Prim’s Algorithm (aka Jarnik’s Algorithm)

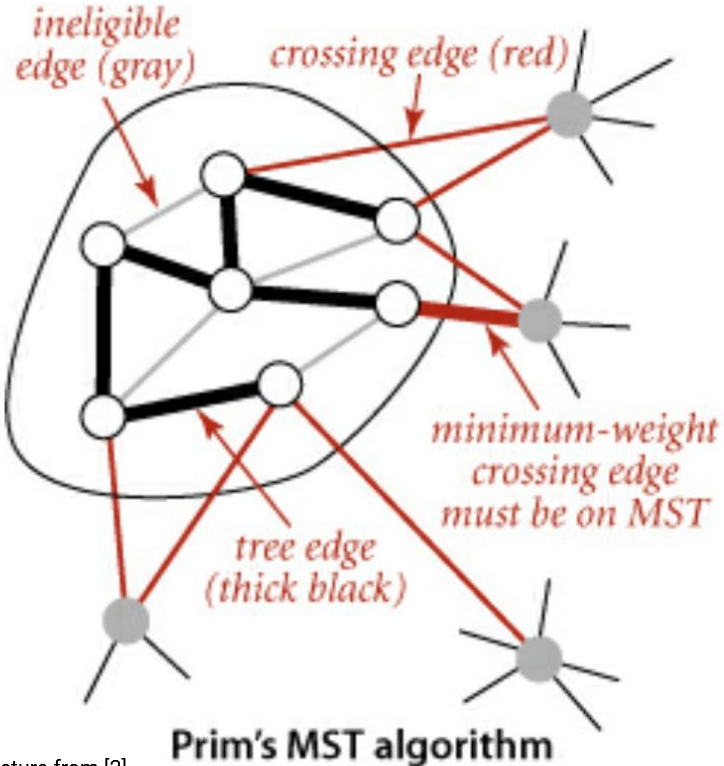
- Grow the tree one edge at a time
- Start with a single vertex (counts as a tree)
- Add |V| - 1 edges to it by adding the minimum-weight edge that connects a new vertex to the tree (crossing the partition that is defined by the current tree vertices--called a *crossing edge*)
- NOTE: When you add an edge to the tree, you are also adding a vertex to the tree.
- Like Dijkstra’s Shortest Path, this is a greedy algorithm.



Picture from [2]

Prim’s Algorithm (aka Jarnik’s Algorithm)

Proof of Correctness: Follows directly from the Partition Property because we are choosing the minimum weight edge across the tree-defined partition.



Picture from [2]

Implementation of Prim’s Algorithm

- How would you implement this algorithm?
- What data structures would be useful?
- What would be the space and runtime requirements?
- What would be the best way to represent the graph?

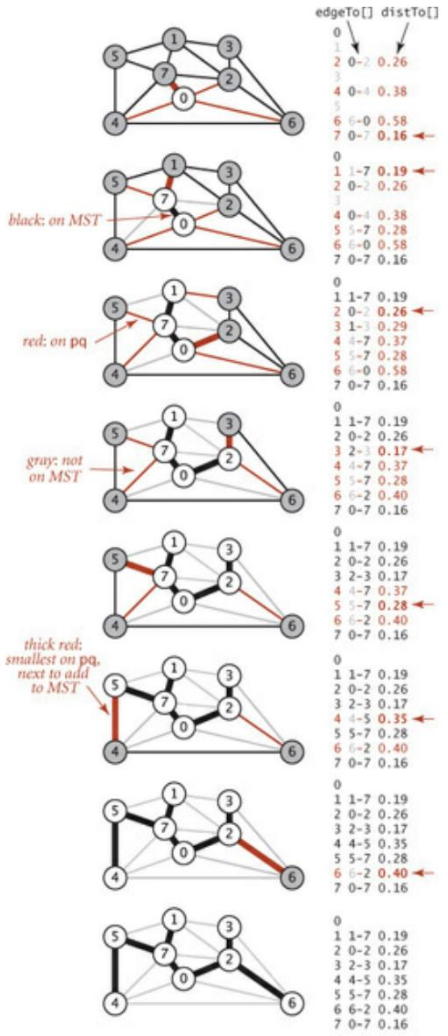
Implementation of Prim’s Algorithm

- **marked[]**: an array of booleans to keep track of vertices on the tree
- **edgeTo[]**: an array to keep track of the lightest edge connecting a new vertex to the tree
- **distTo[]**: an array to keep track of the weight of the lightest edge connecting a new vertex to the tree
- **pq**: a minimum priority queue to keep track of eligible crossing edges (key is the weight of the edges)

Algorithm $PrimsMST(G)$
Input: $G = (V, E)$, a weighted, undirected graph
Output: A minimum-weight spanning tree of G

```
    edgeTo[] := a |V|-sized array to store the edge
        connecting a vertex to the tree
    distTo[] := a |V|-sized array to keep track of
        the distance of the edge connecting a vertex
        to a tree
    marked[] := a |V|-sized array to keep track of
        which vertices have been visited
    pq := a heap-based min priority queue with
        weights as keys and vertices as values

    //initialize structures
    for all v ∈ V:
        distTo[v] = ∞
    distTo[0] = 0
    pq.insert(0, 0)
```

```
//main loop
while !pq.isEmpty:
    v = pq.delMin()
    marked[v] = T
    for each edge e = (v, u) adjacent to v:
        if marked[u]
            continue
        if e.weight() < distTo[u]:
            edgeTo[u] = e
            distTo[u] = e.weight()
            if pq.contains(u):
                pq.changeKey(u, distTo[u])
            else:
                pq.insert(u, distTo[u])
```



Trace of Prim's algorithm (eager version)

Picture from [2]

//main loop

while $!pq.isEmpty$:

$\quad v = pq.delMin()$

$\quad marked[v] = T$

$\quad$ **for each** edge $e = (v, u)$ adjacent to v:

$\quad\quad$ **if** $marked[u]$

$\quad\quad\quad$ **continue**

$\quad\quad$ **if** $e.weight() < distTo[u]$:

$\quad\quad\quad edgeTo[u] = e$

$\quad\quad\quad distTo[u] = e.weight()$

$\quad\quad\quad$ **if** $pq.contains(u)$:

$\quad\quad\quad\quad pq.changeKey(u, distTo[u])$

$\quad\quad\quad$ **else**:

$\quad\quad\quad\quad pq.insert(u, distTo[u])$

Algorithm $PrimsMST(G)$
Input: $G = (V, E)$, a weighted, undirected graph
Output: A minimum-weight spanning tree of G

```
edgeTo[] := a |V|−sized array to store the edge
    connecting a vertex to the tree
distTo[] := a |V|−sized array to keep track of
    the distance of the edge connecting a vertex
    to a tree
marked[] := a |V|−sized array to keep track of
    which vertices have been visited
pq := a heap-based min priority queue with
    weights as keys and vertices as values

//initialize structures
for all v ∈ V:
    distTo[v] = ∞
distTo[0] = 0
pq.insert(0, 0)
```

O(|V|)

```
//main loop
while !pq.isEmpty:
    v = pq.delMin()
    marked[v] = T
    for each edge e = (v, u) adjacent to v:
        if marked[u]
            continue
        if e.weight() < distTo[u]:
            edgeTo[u] = e
            distTo[u] = e.weight()
            if pq.contains(u):
                pq.changeKey(u, distTo[u])
            else:
                pq.insert(u, distTo[u])
```

```
//main loop
while !pq.isEmpty:
    v = pq.delMin()
    marked[v] = T
    for each edge e = (v, u) adjacent to v:
        if marked[u]
            continue
        if e.weight() < distTo[u]:
            edgeTo[u] = e
            distTo[u] = e.weight()
            if pq.contains(u):
                pq.changeKey(u, distTo[u])
            else:
                pq.insert(u, distTo[u])
```

Assuming adjacency list representation, total time is **O((V + E)logV) = O(ElogV)** for a connected graph

- delMin in a heap-based PQ: O(logV)
- checking if PQ contains a vertex and changing a key is O(logV) as long as there is an auxiliary data structure keeping track of positions in the queue
- key of any vertex *v* is updated at most *deg(v)* times and sum of all degrees is O(M)

Imagine...

You are employed at a company that has offices in several different cities. Your first task is to work with a phone company to ensure that there is a connection between every pair of offices according to the following guidelines:

- The connection between two offices does not have to be direct.
- The company wants to minimize the total cost of all the connections.

What kind of a problem is this?

Imagine...

You are employed at a company that has offices in several different cities. Your first task is to work with a phone company to ensure that there is a connection between every pair of offices according to the following specifications:

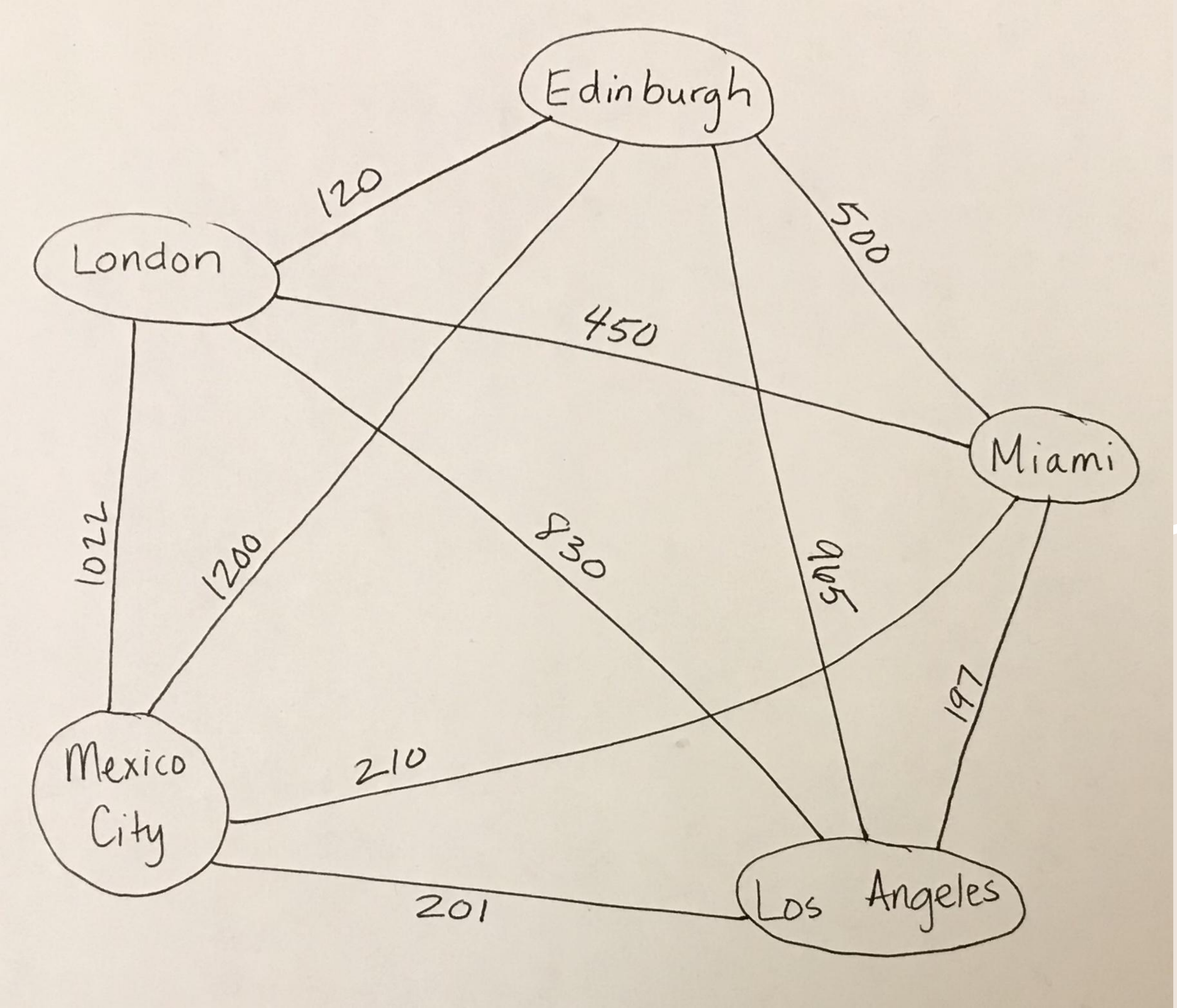
- The phone company charges different rates to connect pairs of cities, specified on the following slide.
- The connection between two offices does not have to be direct.
- The company wants to minimize the total cost of all the connections.

What kind of a problem is this?

A minimum spanning tree problem!

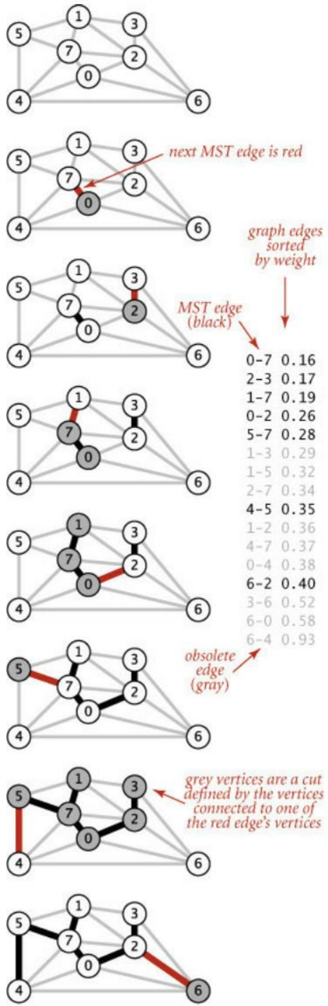
	London	Edinburgh	Miami	Mexico City	Los Angeles
London	0	$120	$450	$1022	$830
Edinburgh	$120	0	$500	$1200	$965
Miami	$450	$500	0	$210	$197
Mexico City	$1022	$1200	$210	0	$201
Los Angeles	$830	$965	$197	$201	0

Goal: Find an MST of the graph using Kruskal's Algorithm...



Kruskal’s Algorithm

- Process edges, adding edges to the tree smallest-weight first **as long as the new edge does not form a cycle.**
- The result is that the algorithm creates a forest that eventually merges into a single tree.
- What would be some of the challenges in implementing this?
- What data structures would be useful?
- How would you represent the graph?



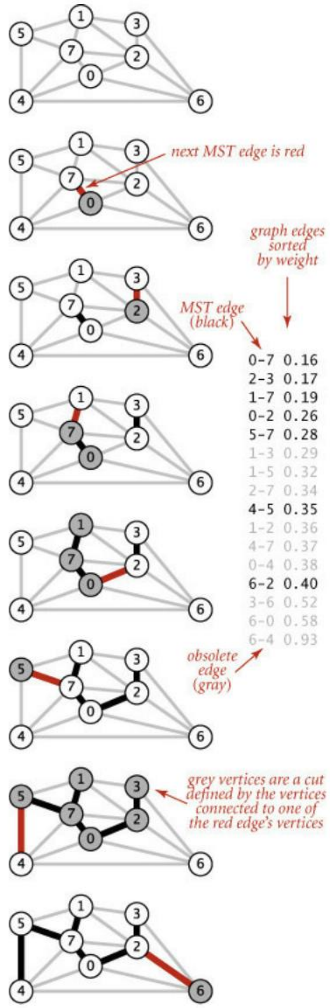
Trace of Kruskal's algorithm

Picture from [2]

Kruskal’s Algorithm

- Can be implemented using disjoint sets for each separate component
- Start with |V| disjoint sets, each containing a single vertex
- Combine sets (union) by adding edges, reducing the number of separate components until there is only one

https://drive.google.com/file/d/1EJpFkGbcih3gT9nS1dVfZFU26_0lIhf3/view?usp=sharing



Trace of Kruskal's algorithm

Picture from [2]

Algorithm $KruskalsMST(G)$

Input: $G = (V, E)$, a connected, weighted, undirected graph

Output: A minimum-weight spanning tree of G

```
pq := a minimum priority queue of edges where keys are weights
for each edge e = (u, v) ∈ E:
    pq.insert(e, e.weight())

A = ∅
for each v ∈ V:
    makeSet(v)
while |A| < |V| − 1
    e = (u, v) = pq.delMin()
    if findSet(u) ≠ findSet(v):
        A = A ∪ (u, v)
        union(u, v)
return A
```

Algorithm $KruskalsMST(G)$
Input: $G = (V, E)$, a connected, weighted, undirected graph
Output: A minimum-weight spanning tree of G

$$
\begin{aligned}
&pq := \text{a minimum priority queue of edges where keys are weights} \\
&\textbf{for each } \text{edge } e = (u, v) \in E: \\
&\quad pq.insert(e, e.weight()) \\
&\\
&A = \emptyset \\
&\textbf{for each } v \in V: \\
&\quad makeSet(v) \\
&\textbf{while } |A| < |V| - 1 \\
&\quad e = (u, v) = pq.delMin() \\
&\quad \textbf{if } findSet(u) \neq findSet(v): \\
&\qquad A = A \cup (u, v) \\
&\qquad union(u, v) \\
&\textbf{return } \text{A}
\end{aligned}
$$

- **O(ElogE)** for doing **E** inserts into the **E**-sized PQ
- Alternatively, we could just sort the edges in **O(ElogE)** time

Algorithm $KruskalsMST(G)$
Input: $G = (V, E)$, a connected, weighted, undirected graph
Output: A minimum-weight spanning tree of G

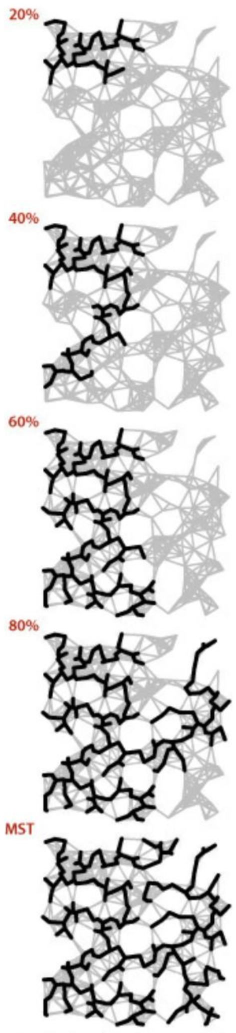
```
pq := a minimum priority queue of edges where keys are weights
for each edge e = (u, v) ∈ E:
    pq.insert(e, e.weight())

A = ∅
for each v ∈ V:
    makeSet(v)
while |A| < |V| − 1
    e = (u, v) = pq.delMin()
    if findSet(u) ≠ findSet(v):
        A = A ∪ (u, v)
        union(u, v)
return A
```

- *makeSet* is an O(1) operation, so the total time is **O(V)**
- The total time required for **E** *union* and *findSet* operations is **O(ElogE)**

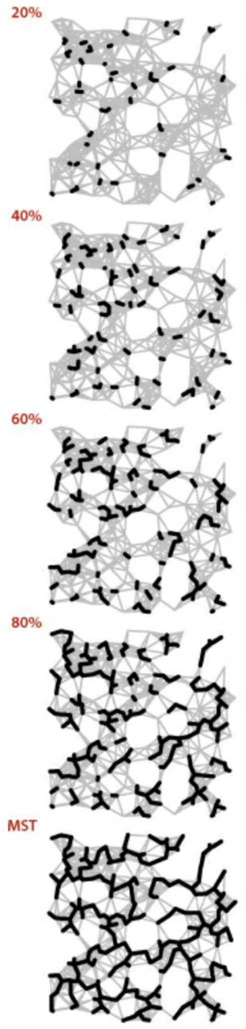
- **Total time: O(ElogE)**

Prim’s vs. Kruskal’s

- Space: |**V**| vs. |**E**|
- Time: |**E**|**log**|**V**| vs. |**E**|**log**|**E**|



Prim’s algorithm (250 vertices)



Kruskal’s algorithm (250 vertices)

Pictures from [2]

Baruvka's Algorithm

- 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).

Picture from [1]

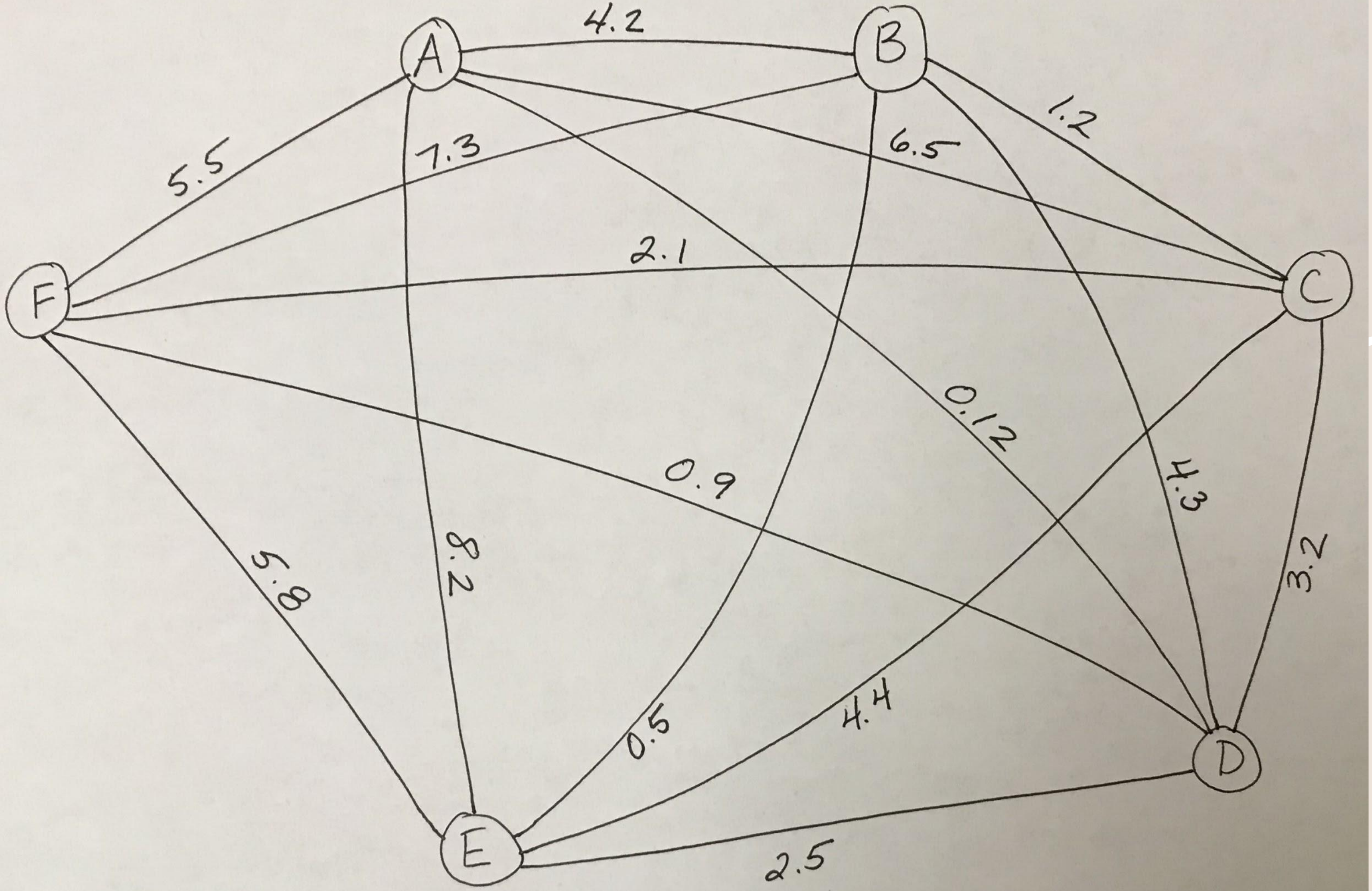
Imagine...

You are employed at a florist shop as a driver during their busy Valentine's Day season. Your first task is to take a list of delivery addresses and plan a route according to the following specifications:

- You want to start and end at the florist shop.
- You want to deliver ALL the flowers
- You want to minimize the distance you have to go.
- You don't want to visit any stop more than once.
- You don't want to drive the same road more than once.

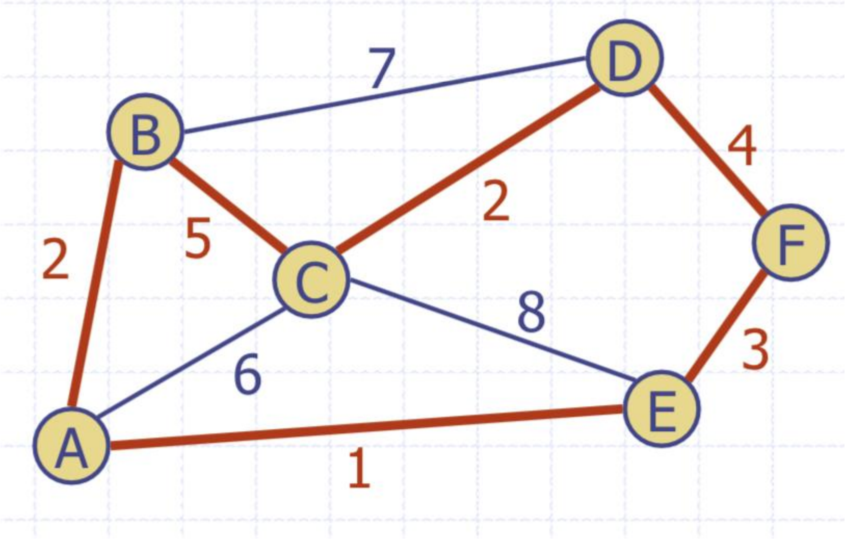
How would you go about solving this problem?

	Florist	A	B	C	D	E
Florist	0	5.5	7.3	2.1	0.9	5.8
A	5.5	0	4.2	6.5	0.12	8.2
B	7.3	4.2	0	1.2	4.3	0.5
C	2.1	6.5	1.2	0	3.2	4.4
D	0.9	0.12	4.3	3.2	0	2.5
E	5.8	8.2	0.5	4.4	2.5	0

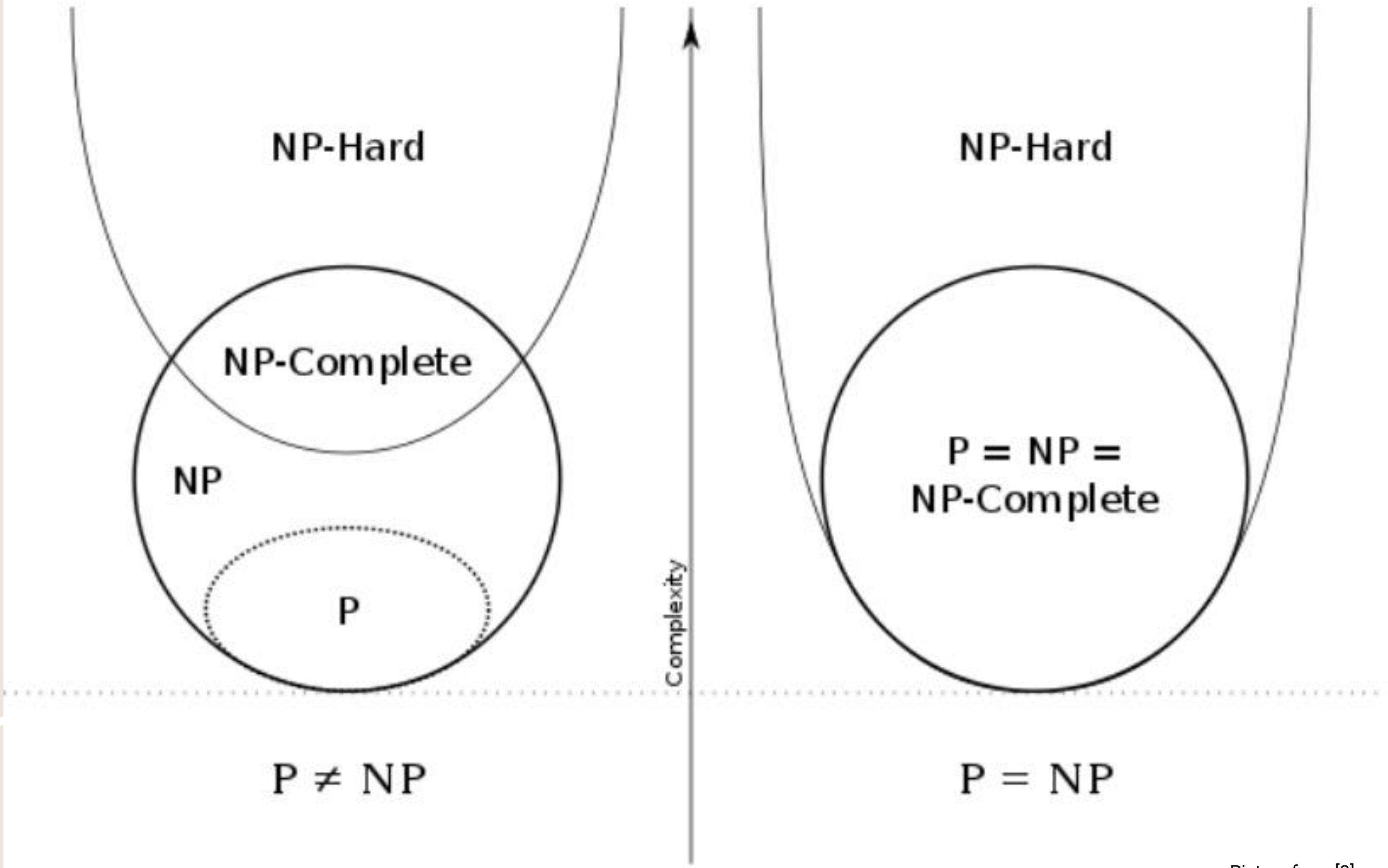

A
B
C
D
E
F
4.2
1.2
6.5
7.3
5.5
2.1
0.12
4.3
0.9
3.2
8.2
5.8
4.4
0.5
2.5

The Traveling Salesperson Problem

- A **tour** of a graph is a spanning cycle (goes through all the vertices)
- A TSP tour of a weighted graph is a tour that is simple (i.e. repeats no vertices or edges) and has minimum weight
- Determining if a TSP tour shorter than a given length L **exists** in a graph is NP-Complete (i.e. solution can be verified in polynomial time but not discovered in polynomial time—at least not yet)
- Finding a TSP tour is NP-hard



Picture from [1]



Picture from [3]

Other NP-Complete Problems

- Boolean Satisfiability Problem (SAT)
- Knapsack Problem
- Hamiltonian Path Problem
- Subgraph Isomorphism Problem
- Subset sum problem
- Clique Problem
- Vertex Cover Problem
- Independent set problem
- Dominating set problem
- Graph Coloring problem

References

[1] Goodrich and Tamassia
[2] Sedgewick and Wayne
[3] en.wikipedia.org