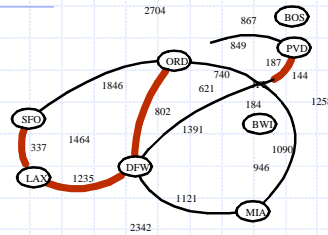


Minimum Spanning Trees



Minimum Spanning Trees

Spanning subgraph

- Subgraph of a graph G containing all the vertices of G

Spanning tree

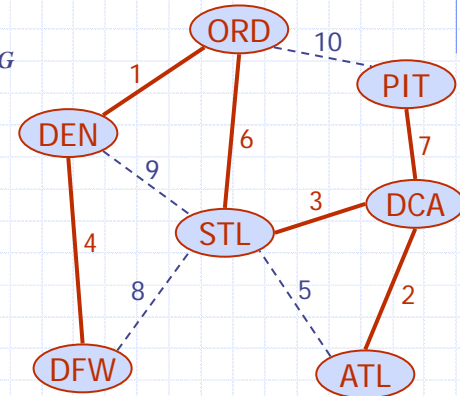
- Spanning subgraph that is itself a (free) tree

Minimum spanning tree (MST)

- Spanning tree of a weighted graph with minimum total edge weight

Applications

- Communications networks
- Transportation networks



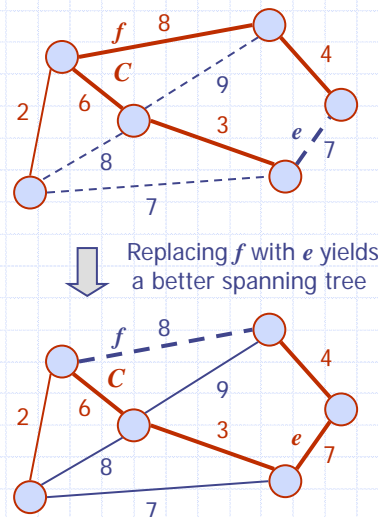
Cycle Property

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 , $\text{weight}(f) \leq \text{weight}(e)$

Proof:

- By contradiction
- If $\text{weight}(f) > \text{weight}(e)$ we can get a spanning tree of smaller weight by replacing e with f



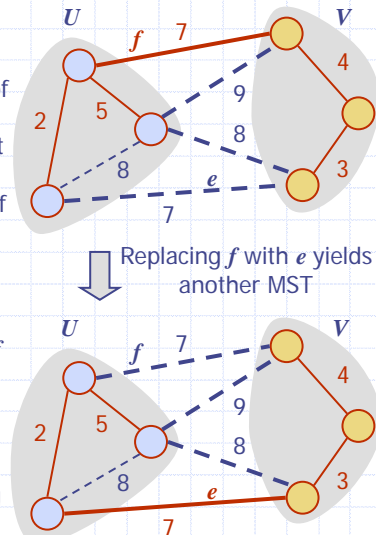
Partition Property

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, $\text{weight}(f) \leq \text{weight}(e)$
- Thus, $\text{weight}(f) = \text{weight}(e)$
- We obtain another MST by replacing f with e

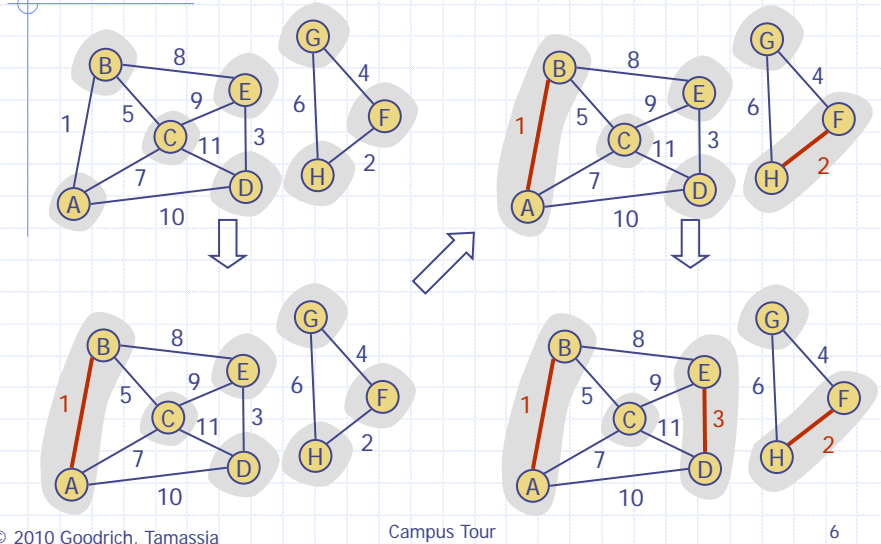


Kruskal's Algorithm

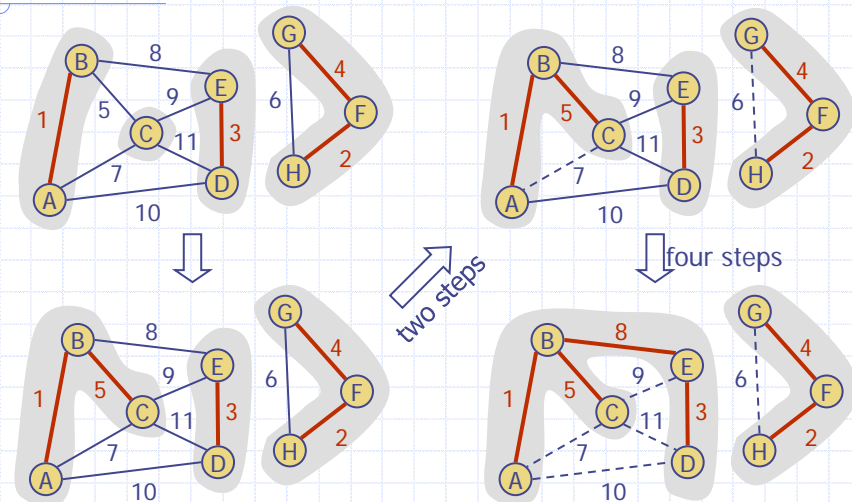
- ❑ Maintain a partition of the vertices into clusters
 - Initially, single-vertex clusters
 - Keep an MST for each cluster
 - Merge "closest" clusters and their MSTs
- ❑ A priority queue stores the edges outside clusters
 - Key: weight
 - Element: edge
- ❑ At the end of the algorithm
 - One cluster and one MST

Algorithm *KruskalMST(G)*
for each vertex v in G **do**
 Create a cluster consisting of v
let Q be a priority queue.
 Insert all edges into Q
 $T \leftarrow \emptyset$
 { T is the union of the MSTs of the clusters }
while T has fewer than $n - 1$ edges **do**
 $e \leftarrow Q.\text{removeMin}().\text{getValue}()$
 $[u, v] \leftarrow G.\text{endVertices}(e)$
 $A \leftarrow \text{getCluster}(u)$
 $B \leftarrow \text{getCluster}(v)$
 if $A \neq B$ **then**
 Add edge e to T
 mergeClusters(A, B)
return T

Example



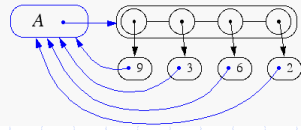
Example (contd.)



Data Structure for Kruskal's Algorithm

- ❑ The algorithm maintains a forest of trees
- ❑ A priority queue extracts the edges by increasing weight
- ❑ An edge is accepted if it connects distinct trees
- ❑ We need a data structure that maintains a **partition**, i.e., a collection of disjoint sets, with operations:
 - **makeSet**(u): create a set consisting of u
 - **find**(u): return the set storing u
 - **union**(A, B): replace sets A and B with their union

Recall of List-based 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**(A,B), we move the elements of the smaller set to the sequence of the larger set and update their references
 - the time for operation **union**(A,B) is $\min(|A|, |B|)$
- 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

Partition-Based Implementation

- Partition-based version of Kruskal's Algorithm
 - Cluster merges as unions
 - Cluster locations as finds
- Running time $O((n + m) \log n)$
 - PQ operations $O(m \log n)$
 - UF operations $O(n \log n)$

Algorithm **KruskalMST**(G)

Initialize a partition **P**

for each vertex **v** in **G** do

P.makeSet(v)

let **Q** be a priority queue.

Insert all edges into **Q**

T ← ∅

{**T** is the union of the MSTs of the clusters}

while **T** has fewer than $n - 1$ edges do

e ← **Q.removeMin().getValue()**

[**u**, **v**] ← **G.endVertices**(e)

A ← **P.find**(u)

B ← **P.find**(v)

if **A** ≠ **B** then

Add edge **e** to **T**

P.union(**A**, **B**)

return **T**

Prim-Jarnik's Algorithm

- Similar to Dijkstra's algorithm
- 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** label **d**(v) representing 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-Jarnik's Algorithm (cont.)

- A heap-based adaptable priority queue with location-aware entries stores the vertices outside the cloud
 - Key: distance
 - Value: vertex
 - Recall that method **replaceKey**(**l**,**k**) changes the key of entry **l**
- We store three labels with each vertex:
 - Distance
 - Parent edge in MST
 - Entry in priority queue

Algorithm **PrimJarnikMST**(G)

Q ← new heap-based priority queue

s ← a vertex of **G**

for all **v** ∈ **G.vertices**()

if **v** = **s**

v.setDistance(0)

else

v.setDistance(∞)

v.setParent(∅)

l ← **Q.insert**(**v.getDistance**(), **v**)

v.setLocator(**l**)

while ¬**Q.empty**()

l ← **Q.removeMin**()

u ← **l.getValue**()

for all **e** ∈ **u.incidentEdges**()

z ← **e.opposite**(**u**)

r ← **e.weight**()

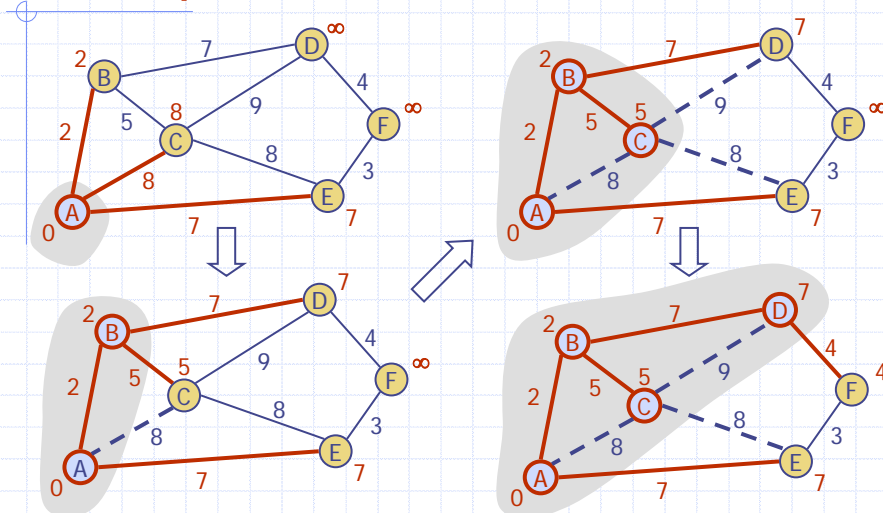
if **r** < **z.getDistance**()

z.setDistance(**r**)

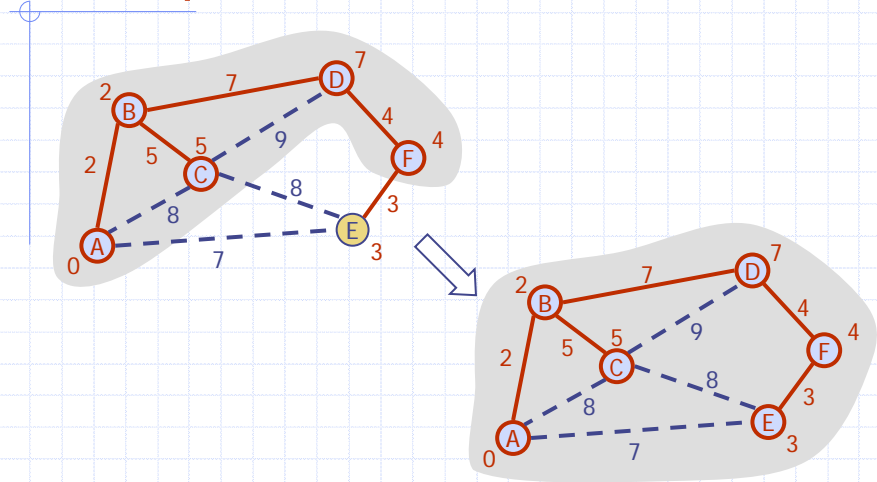
z.setParent(**e**)

Q.replaceKey(**z.getEntry**(), **r**)

Example



Example (contd.)



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 (Exercise)

- Like Kruskal's Algorithm, Baruvka's algorithm grows many clusters at once and maintains a forest T
- Each iteration of the while loop halves the number of connected components in forest T
- The running time is $O(m \log n)$

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
    
```

Example of Baruvka's Algorithm (animated)

Slide by Matt Stallmann
included with permission.

