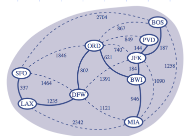
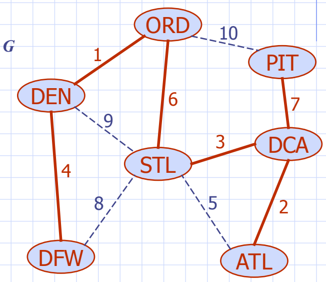
Minimum Spanning Trees



Minimum Spanning Trees

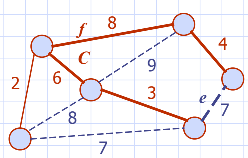
- Spanning subgraph
  - Subgraph of a graph G containing all the vertices of G
- Spanning tree
  - Spanning subgraph that is itself a (free) tree
- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight
- Applications
  - Communications networks
  - Transportation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



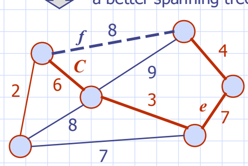
Cycle Property

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



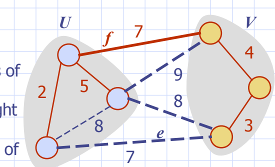
Replacing f with e yields a better spanning tree



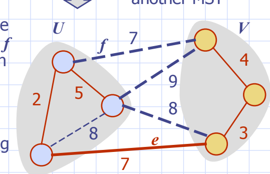
Partition Property

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



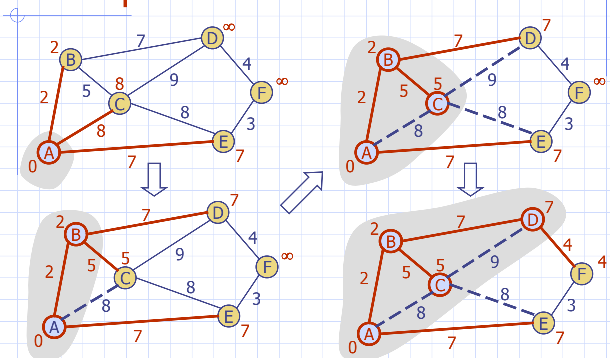
Prim-Jarnik's Algorithm

- 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

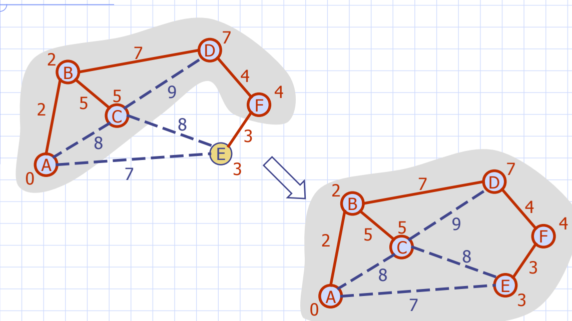
Prim-Jarnik Pseudo-code

Algorithm PrimJarnik(G):
  Input: An undirected, weighted, connected graph G with n vertices and m edges
  Output: A minimum spanning tree T for G
Pick any vertex s of G
$D[s] = 0$
for each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
while Q is not empty **do**
  $(u, e) =$ value returned by Q.remove_min()
  Connect vertex u to T using edge e.
  for each edge $e' = (u, v)$ such that v is in Q **do**
    {check if edge (u, v) better connects v to T}
    if $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex v in Q to $D[v]$.
      Change the value of vertex v in Q to (v, e').
return the tree T

Example



Example (contd.)



Analysis

- Graph operations
  - We cycle through the incident edges 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 $\Sigma_v \deg(v) = 2m$
- The running time is $O(m \log n)$ since the graph is connected

Python Implementation

```
def MST_PrimJarnik(g):
  """Compute a minimum spanning tree of weighted graph g.

  Return a list of edges that comprise the MST (in arbitrary order).
  """
  d = { }                              # d[v] is bound on distance to tree
  tree = [ ]                           # list of edges in spanning tree
  pq = AdaptableHeapPriorityQueue( )   # d[v] maps to value (v, e=(u,v))
  pqlocator = { }                      # map from vertex to its pq locator

  # for each vertex v of the graph, add an entry to the priority queue, with
  # the source having distance 0 and all others having infinite distance
  for v in g.vertices( ):
    if len(d) == 0:                    # this is the first node
      d[v] = 0                         # make it the root
    else:
      d[v] = float('inf')              # positive infinity
    pqlocator[v] = pq.add(d[v], (v,None))

  while not pq.is_empty( ):
    key,value = pq.remove_min( )
    u,edge = value                     # unpack tuple from pq
    del pqlocator[u]                   # u is no longer in pq
    if edge is not None:
      tree.append(edge)                # add edge to tree
    for link in g.incident_edges(u):
      v = link.opposite(u)
      if v in pqlocator:               # thus v not yet in tree
        # see if edge (u,v) better connects v to the growing tree
        wgt = link.element( )
        if wgt < d[v]:                 # better edge to v?
          d[v] = wgt                   # update the distance
          pq.update(pqlocator[v], d[v], (v, link))   # update the pq entry
  return tree
```

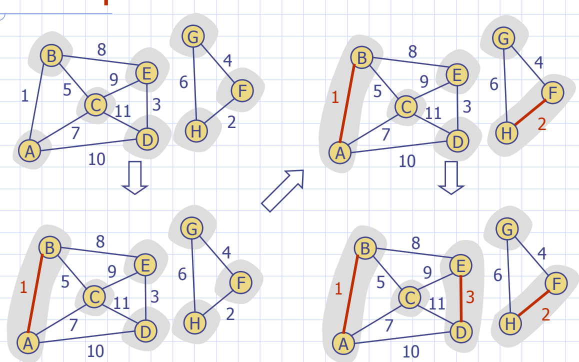

Kruskal's Approach

- 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

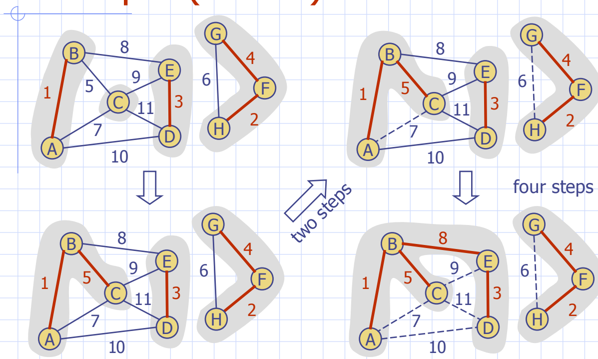
Kruskal's Algorithm

Algorithm Kruskal(G):
Input: A simple connected weighted graph G with n vertices and m edges
Output: A minimum spanning tree T for G
for each vertex v in G **do**
  Define an elementary cluster $C(v) = \{v\}$.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
$T = \emptyset$ $\{T$ will ultimately contain the edges of the MST$\}$
while T has fewer than $n-1$ edges **do**
  (u, v) = value returned by Q.remove_min()
  Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.
  if $C(u) \neq C(v)$ **then**
    Add edge (u, v) to T.
    Merge $C(u)$ and $C(v)$ into one cluster.
return tree T

Example



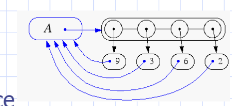
Example (contd.)



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 it if 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

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 $O((n + m) \log n)$
  - Priority Queue operations: $O(m \log n)$
  - Union-Find operations: $O(n \log n)$

Python Implementation

```
def MST_Kruskal(g):
  """Compute a minimum spanning tree of a graph using Kruskal's algorithm.

  Return a list of edges that comprise the MST.

  The elements of the graph's edges are assumed to be weights.
  """
  tree = [ ]                    # list of edges in spanning tree
  pq = HeapPriorityQueue( )     # entries are edges in G, with weights as key
  forest = Partition( )         # keeps track of forest clusters
  position = { }                # map each node to its Partition entry

  for v in g.vertices( ):
    position[v] = forest.make_group(v)

  for e in g.edges( ):
    pq.add(e.element( ), e)     # edge's element is assumed to be its weight

  size = g.vertex_count( )
  while len(tree) != size − 1 and not pq.is_empty( ):
    # tree not spanning and unprocessed edges remain
    weight,edge = pq.remove_min( )
    u,v = edge.endpoints( )
    a = forest.find(position[u])
    b = forest.find(position[v])
    if a != b:
      tree.append(edge)
      forest.union(a,b)

  return tree
```


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)
  $T \leftarrow V$ {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.