Minimum Spanning Tree

Outline and Reading

- Minimum Spanning Trees (§7.3)
  - Definitions
  - A crucial fact
- Prim-Jarnik's Algorithm (§7.3.2)
- Kruskal's Algorithm (§7.3.1)

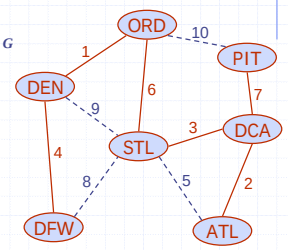
Minimum Spanning Tree

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

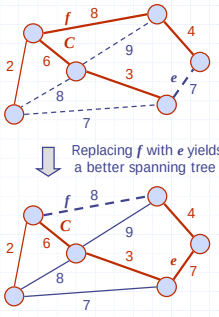
Applications
- Communications networks
- Transportation networks



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, $weight(f) \leq weight(e)$

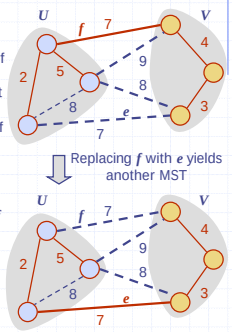
Proof:
- By contradiction
- If $weight(f) > weight(e)$ we can get a spanning tree of smaller weight by replacing e with f



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



Prim-Jarnik's Algorithm

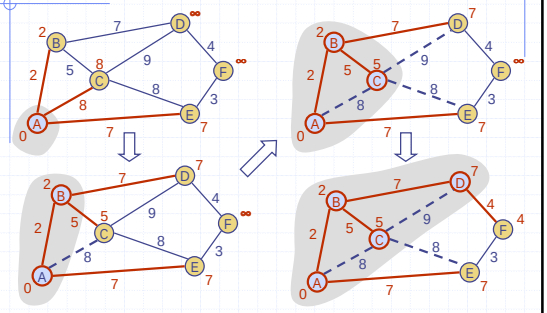
- Prim-Jarnik's algorithm for computing an MST is similar to Dijkstra's algorithm
- We assume that the graph is connected
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v a label $d(v)$ representing the smallest weight of an edge 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.)

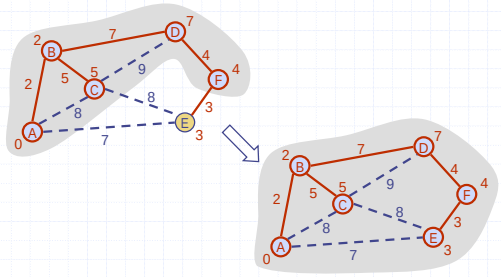
- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - *insert*(*k*,*e*) returns a locator
  - *replaceKey*(*l*,*k*) changes the key of an item
- We store three labels with each vertex:
  - Distance
  - Parent edge in MST
  - Locator 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*
    setDistance(*v*, 0)
  else
    setDistance(*v*, ∞)
  setParent(*v*, ∅)
  l ← *Q.insert*(*getDistance*(*v*), *v*)
  setLocator(*v*,*l*)
while ¬*Q.isEmpty*()
  u ← *Q.removeMin*()
  for all *e* ∈ *G.incidentEdges*(*u*)
    z ← *G.opposite*(*u*,*e*)
    r ← *weight*(*e*)
    if *r* < *getDistance*(*z*)
      setDistance(*z*,*r*)
      setParent(*z*,*e*)
      Q.replaceKey(*getLocator*(*z*),*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

Dijkstra vs. Prim-Jarnik

Algorithm *DijkstraShortestPaths*(*G*, *s*)
Q ← new heap-based priority queue
for all *v* ∈ *G.vertices*()
  if *v* = *s*
    setDistance(*v*, 0)
  else
    setDistance(*v*, ∞)
  setParent(*v*, ∅)
  l ← *Q.insert*(*getDistance*(*v*), *v*)
  setLocator(*v*,*l*)
while ¬*Q.isEmpty*()
  u ← *Q.removeMin*()
  for all *e* ∈ *G.incidentEdges*(*u*)
    z ← *G.opposite*(*u*,*e*)
    r ← *getDistance*(*u*) + *weight*(*e*)
    if *r* < *getDistance*(*z*)
      setDistance(*z*,*r*)
      setParent(*z*,*e*)
      Q.replaceKey(*getLocator*(*z*),*r*)

Algorithm *PrimJarnikMST*(*G*)
Q ← new heap-based priority queue
s ← a vertex of *G*
for all *v* ∈ *G.vertices*()
  if *v* = *s*
    setDistance(*v*, 0)
  else
    setDistance(*v*, ∞)
  setParent(*v*, ∅)
  l ← *Q.insert*(*getDistance*(*v*), *v*)
  setLocator(*v*,*l*)
while ¬*Q.isEmpty*()
  u ← *Q.removeMin*()
  for all *e* ∈ *G.incidentEdges*(*u*)
    z ← *G.opposite*(*u*,*e*)
    r ← *weight*(*e*)
    if *r* < *getDistance*(*z*)
      setDistance(*z*,*r*)
      setParent(*z*,*e*)
      Q.replaceKey(*getLocator*(*z*),*r*)

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*) **then**
    Add edge *e* to *T*
    Merge *Cloud*(*v*) and *Cloud*(*u*)
return *T*