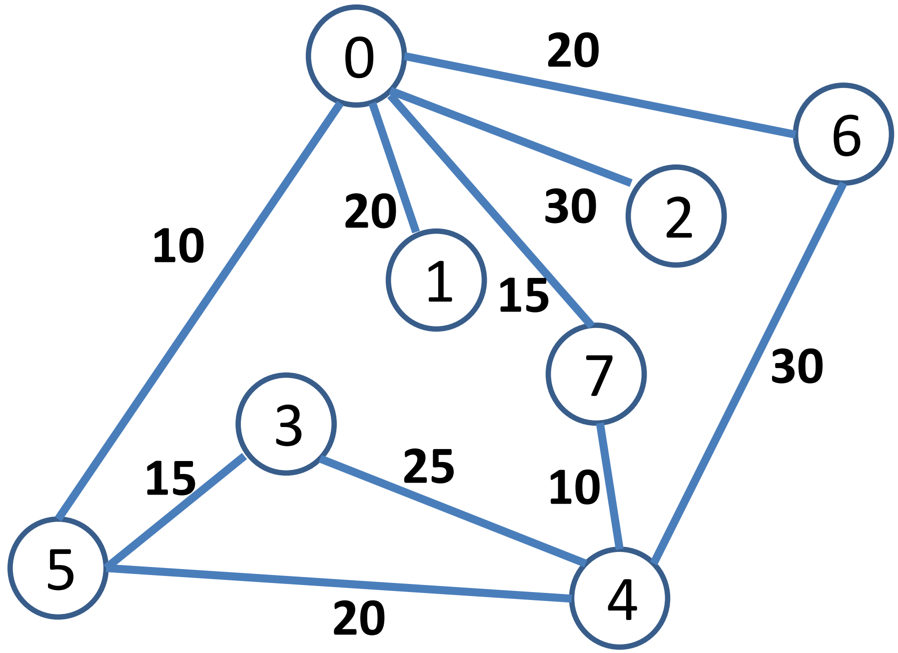
Minimum Spanning Trees

CSE 2320 – Algorithms and Data Structures
Vassilis Athitsos
University of Texas at Arlington

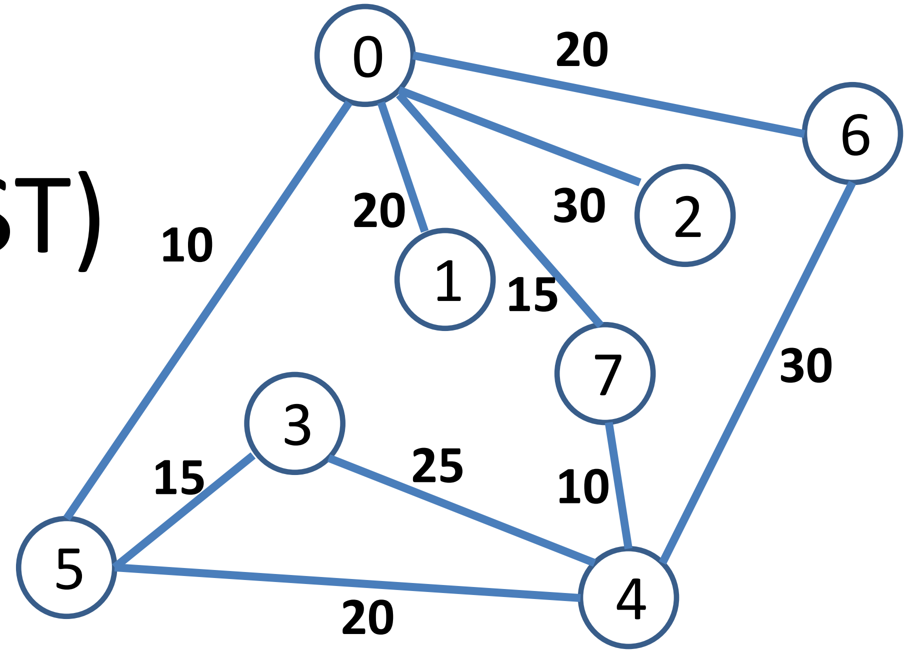
Weighted Graphs

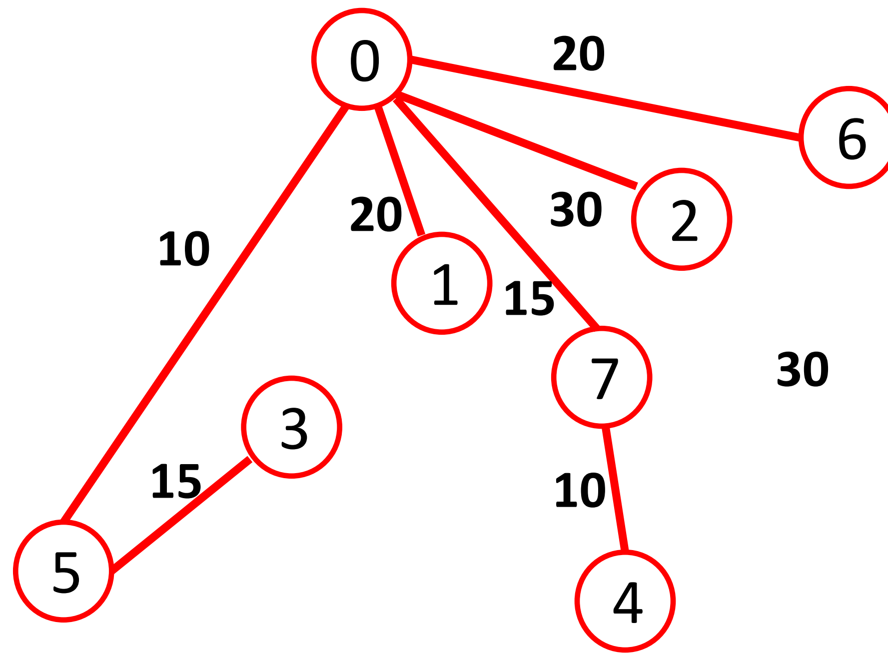
- Each edge has a weight.
- Example: a transportation network (roads, railroads, subway). The weight of each road can be:
  - The length.
  - The expected time to traverse.
  - The expected cost to build.
- Example: a computer network, the weight of each edge (direct link) can be:
  - Latency.
  - Expected cost to build.



Minimum-Cost Spanning Tree (MST)

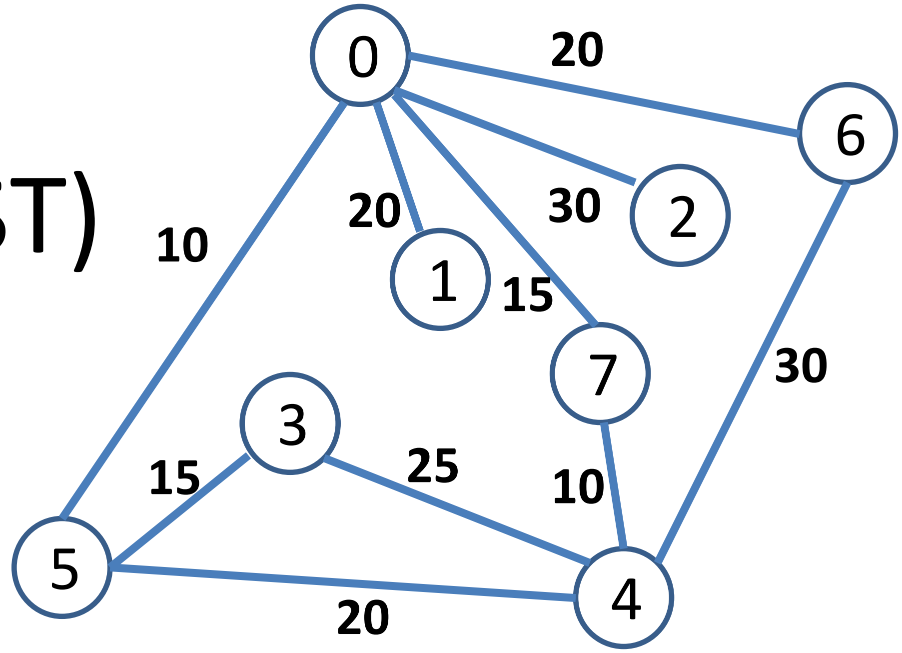
- Important problem in weighted graphs: finding a minimum-cost spanning tree:
- A tree that:
  - Connects **<u>all</u>** vertices of the graph.
  - Has the smallest possible total weight of edges.

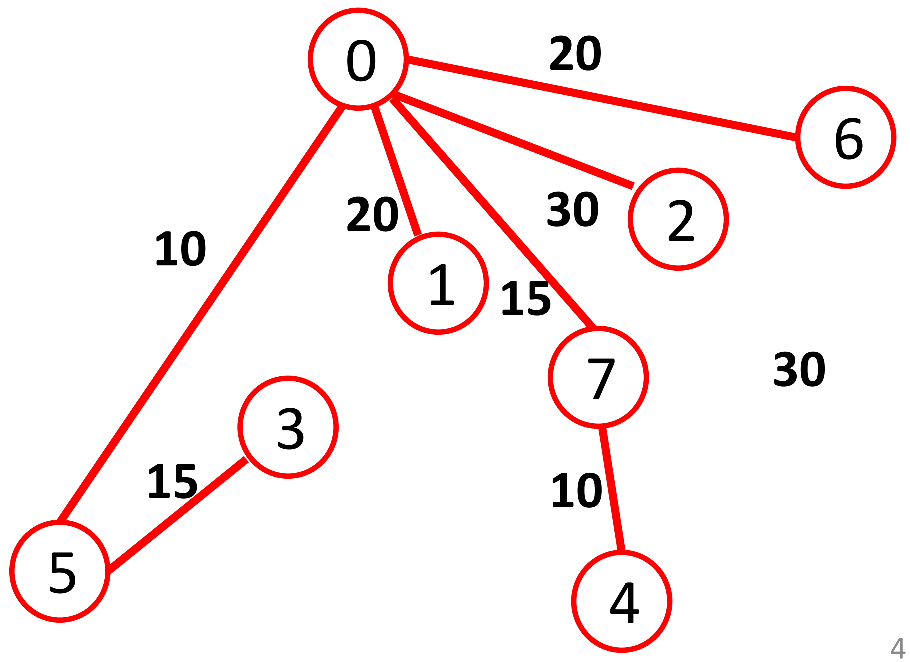




Minimum-Cost Spanning Tree (MST)

- We will only consider algorithms that compute the MST for **undirected graphs**.
- We will allow edges to have negative weights.
- Warning: later in the course (when we discuss Dijkstra's algorithm) we will need to make opposite assumptions:
  - Allow directed graphs.
  - Not allow negative weights.





Prim's Algorithm - Overview

- Prim's algorithm:
  - Start from an tree that contains a single vertex.
  - Keep growing that tree, by adding at each step the shortest edge connecting a vertex in the tree to a vertex outside the tree.
- As you see, it is a very simple algorithm, when stated abstractly.
- However, we have several choices regarding how to implement this algorithm.
- We will see three implementations, with **significantly different properties** from each other.

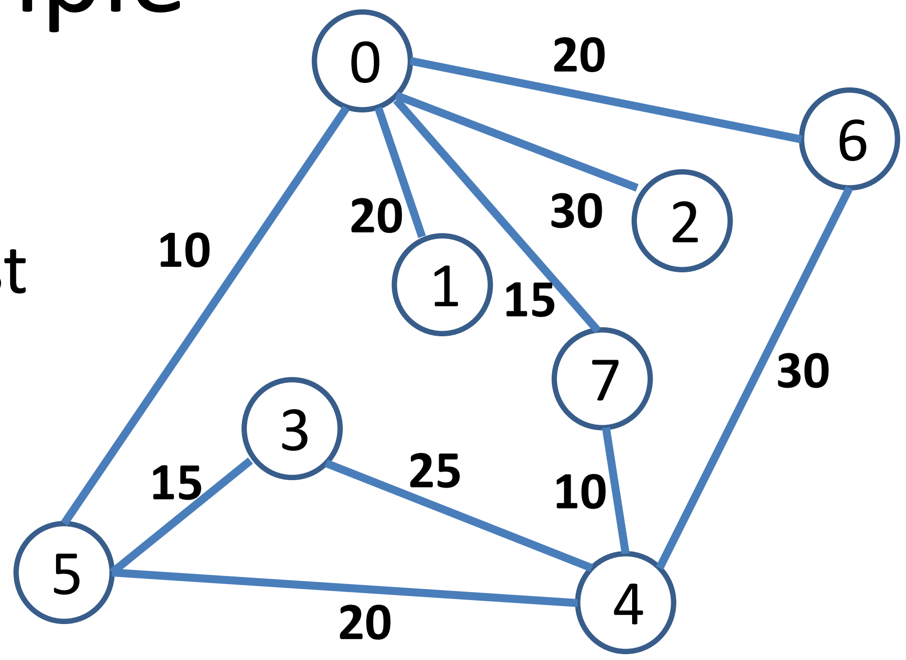
Prim's Algorithm - Simple Version

- Assume an adjacency matrix representation.
  - Each vertex is a number from 0 to V-1.
  - We have a V*V adjacency matrix ADJ, where: ADJ[v][w] is the weight of the edge connecting v and w.
  - If v and w are not connected, ADJ[v][w] = infinity.

Prim's Algorithm - Simple Version

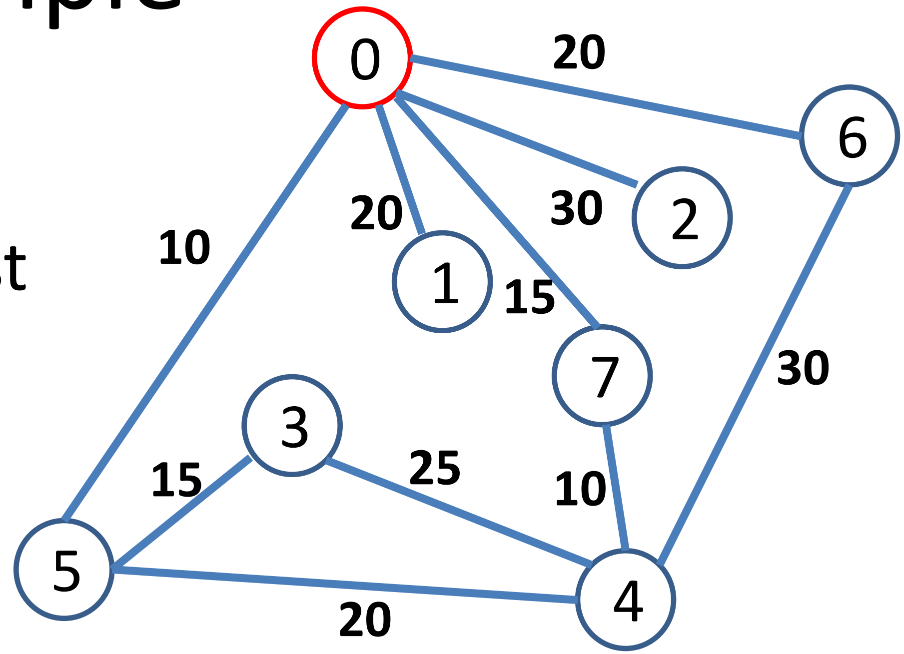
1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
    3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
    4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.

Example



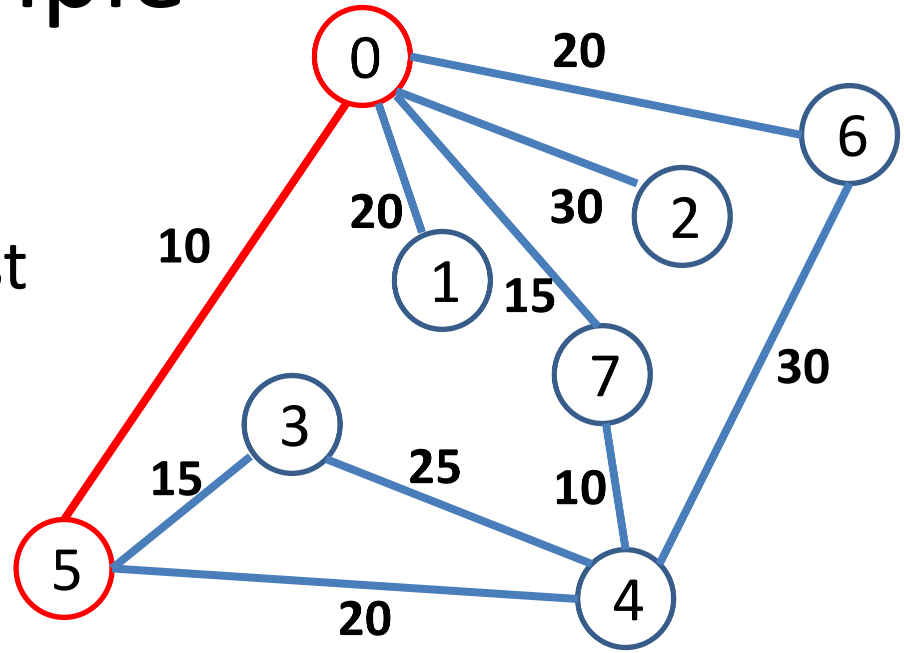
1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
   3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
   4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.

Example



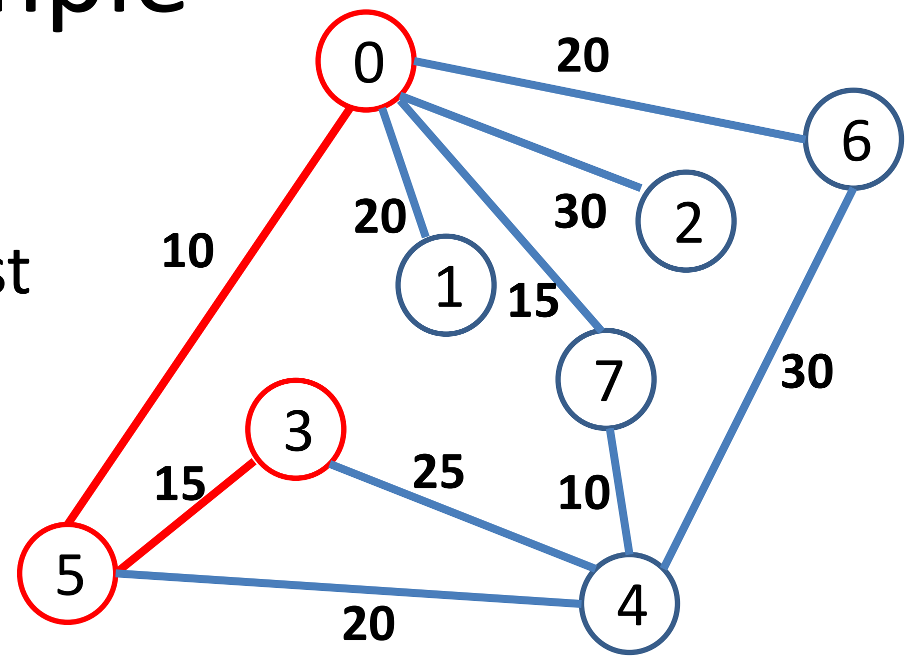
1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
   3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
   4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.

Example



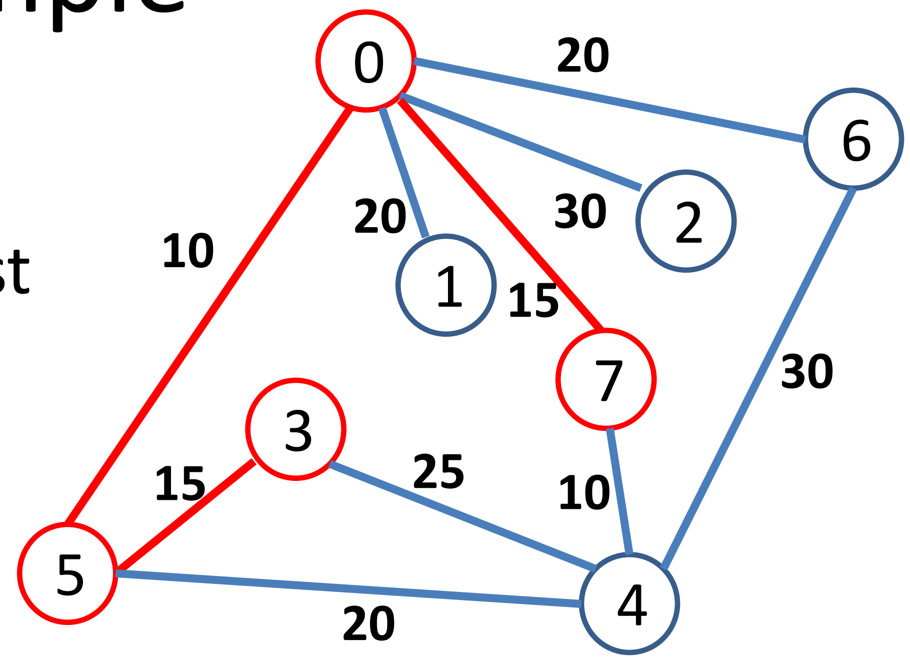
1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
   3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
   4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.

Example



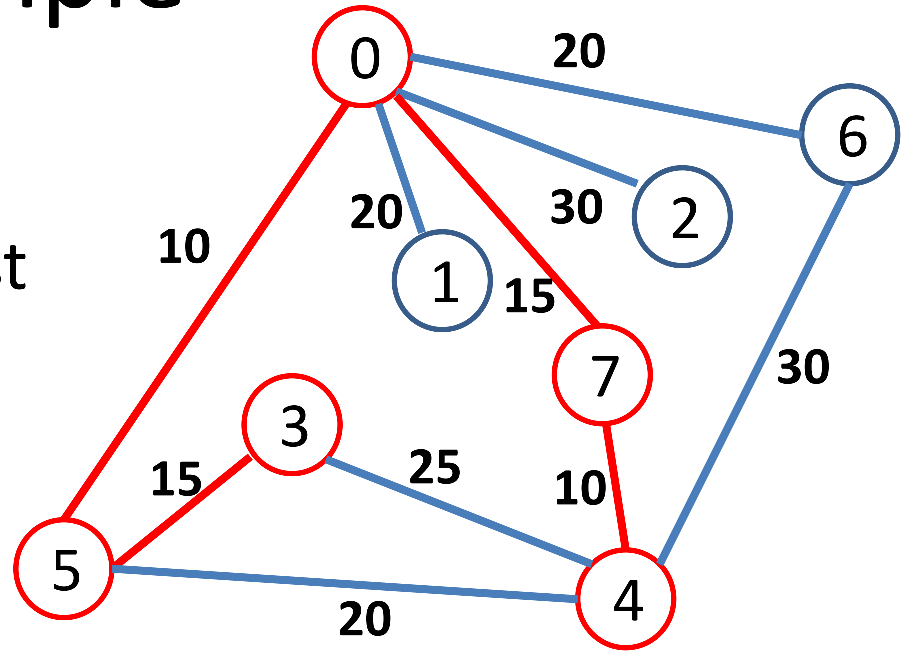
1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
   3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
   4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.

Example



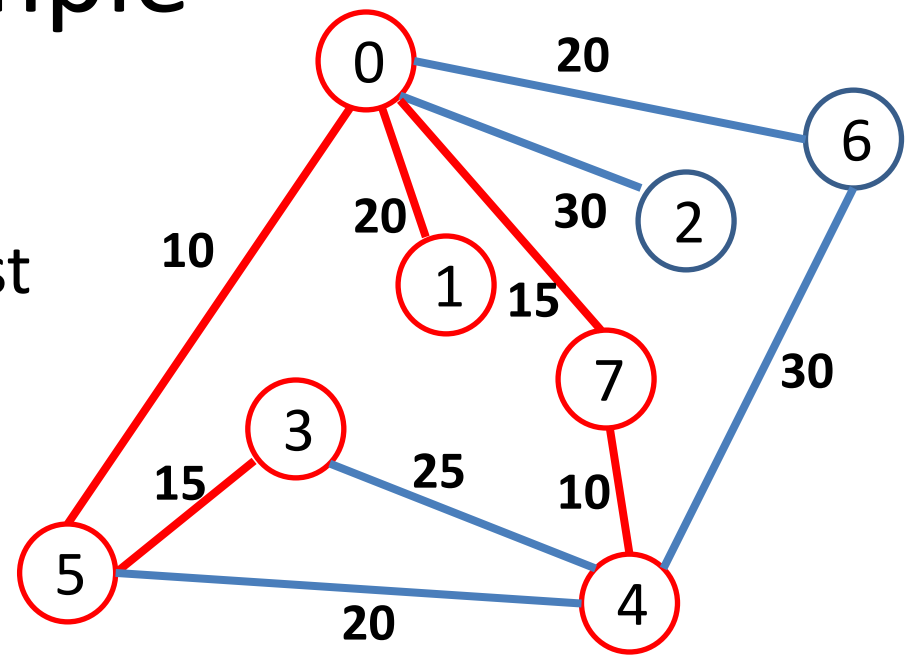
1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
    3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
    4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.

Example



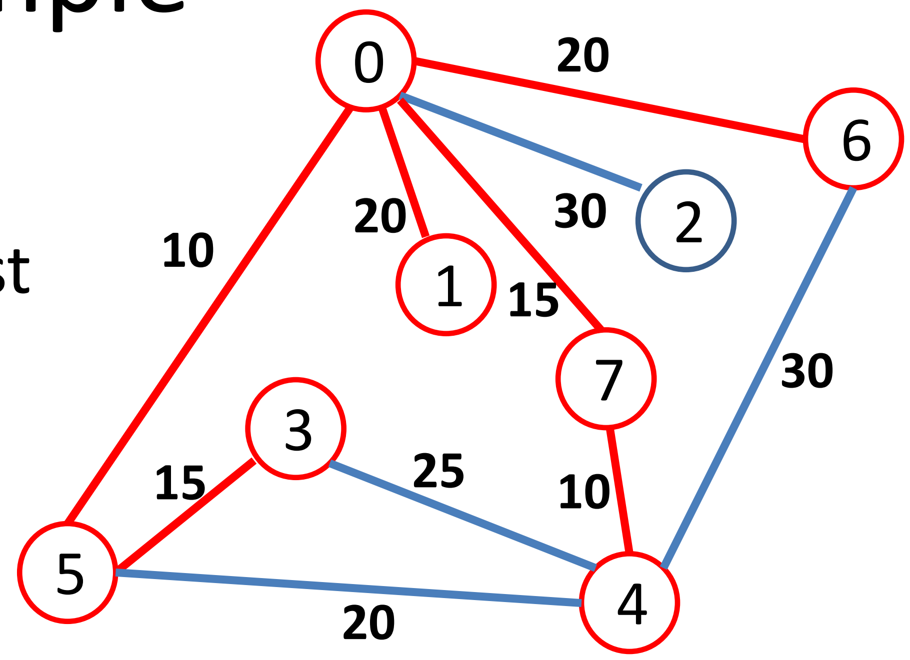
1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
    3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
    4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.

Example



1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
    3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
    4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.

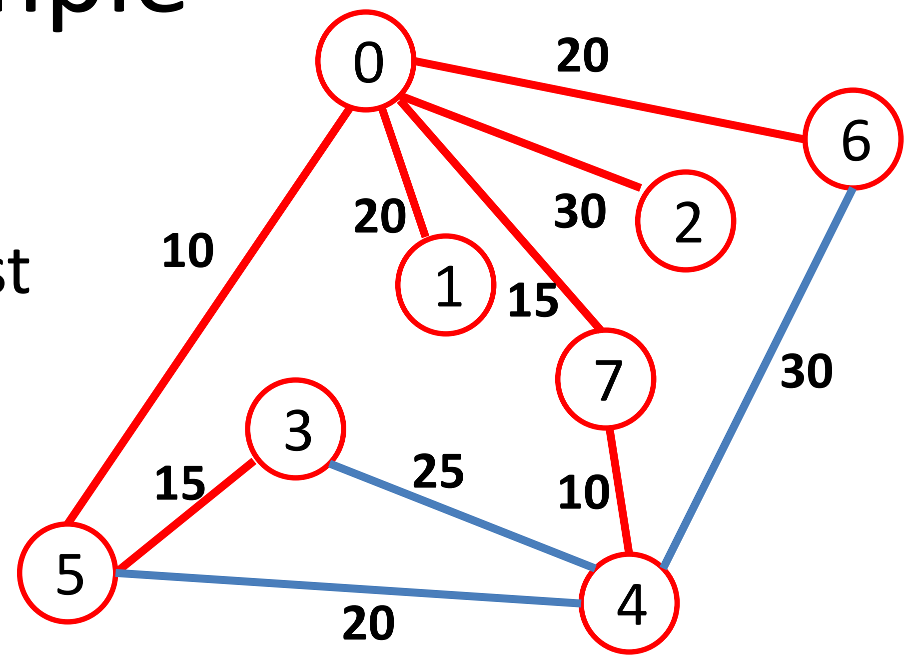
Example



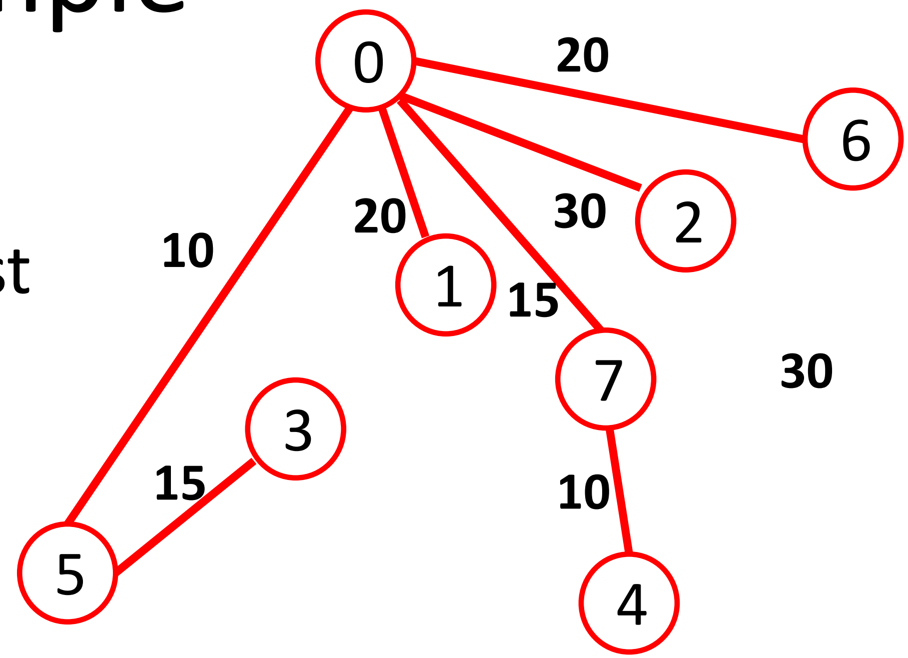
1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
   3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
   4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.

Example

1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
   3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
   4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.



Example



1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
    3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
    4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.

Prim's Algorithm - Simple Version

1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
    3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
    4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.

- Running time?

Prim's Algorithm - Simple Version

1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
   3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
   4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.

- Most naive implementation: time ???
  - Every time we add a new vertex and edge, go through all edges again, to identify the next edge (and vertex) to add.

Prim's Algorithm - Simple Version

1. Start by adding vertex 0 to the MST (minimum-cost spanning tree).
2. Repeat until all vertices have been added to the tree:
   3. From all edges connecting vertices from the current tree to vertices outside the current tree, select the smallest edge.
   4. Add that edge to the tree, and also add to the tree the non-tree vertex of that edge.

- Most naive implementation: time O(VE).
  – Every time we add a new vertex and edge, go through all edges again, to identify the next edge (and vertex) to add.

Prim's Algorithm - Dense Graphs

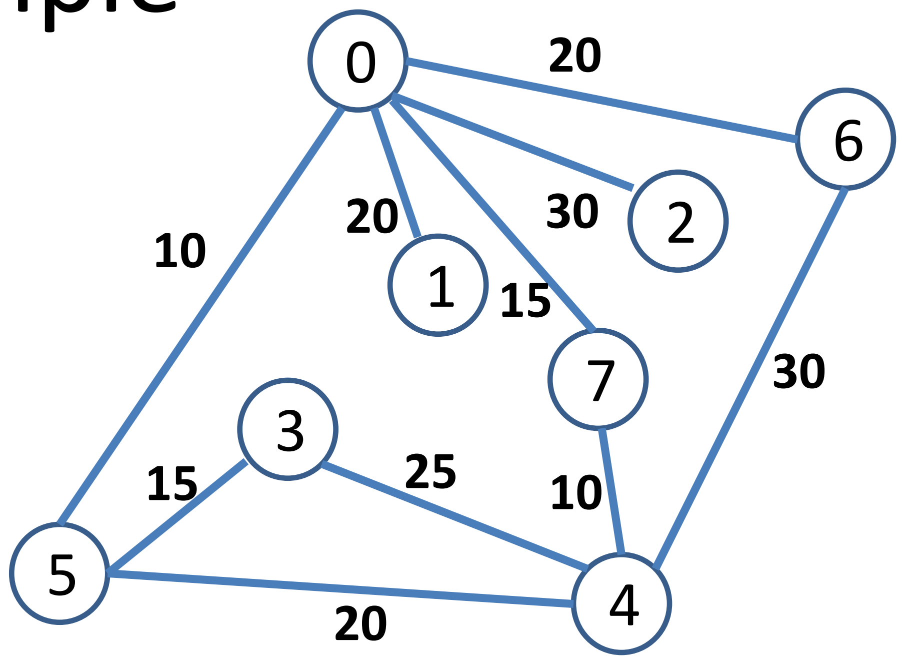
- A dense graph is nearly full, and thus has $O(V^2)$ edges.
  - For example, think of a graph where each vertex has at least V/2 neighbors.
- Just reading the input (i.e., looking at each edge of the graph once) takes $O(V^2)$ time.
- Thus, we cannot possibly compute a minimum-cost spanning tree for a dense graph in less than $O(V^2)$ time.
- Prim's algorithm can be implemented so as to take O(V2) time, which is optimal for dense graphs.

Prim's Algorithm - Dense Graphs

- Again, assume an **adjacency matrix** representation.
- Every time we add a vertex to the MST, we need to update, for each vertex W not in the tree:
  - The smallest edge wt[W] connecting it to the tree.
  - If no edge connects W to the tree, wt[W] = infinity.
  - The tree vertex fr[W] associated with the edge whose weight is wt[W].
- These quantities can be updated in O(V) time when adding a new vertex to the tree.
- Then, the next vertex to add is the one with the smallest wt[W].

Example

- Every time we add a vertex to the MST, we need to update, for each vertex W not in the tree:
  - The smallest edge wt[W] connecting it to the tree.
  - If no edge connects W to the tree, wt[W] = infinity.
  - The tree vertex fr[W] associated with the edge whose weight is wt[W].



Prim's Algorithm: Dense Graphs

```
#define P G->adj[v][w]
void GRAPHmstV(Graph G, int st[], double wt[])
 { int v, w, min;
   for (v = 0; v < G->V; v++)
     { st[v] = -1; fr[v] = v; wt[v] = maxWT; }
   st[0] = 0; wt[G->V] = maxWT;
   for (min = 0; min != G->V; )
     {
       v = min; st[min] = fr[min];
       for (w = 0, min = G->V; w < G->V; w++)
         if (st[w] == -1)
           {
             if (P < wt[w])
               { wt[w] = P; fr[w] = v; }
             if (wt[w] < wt[min]) min = w;
           }}}
```

Prim's Algorithm - Dense Graphs

- Running time: ???

Prim's Algorithm - Dense Graphs

- Running time: $O(V^2)$
- Optimal for **dense graphs**.

Prim's Algorithm for Sparse Graphs

- A sparse graph is one that is not dense.
- This is somewhat vague.
- If you want a specific example, think of a case where the number of edges is linear to the number of vertices.
  - For example, if each vertex can only have between 1 and 10 neighbors, than the number of edges can be at most ???

Prim's Algorithm for Sparse Graphs

- A sparse graph is "one that is not dense".
- This is somewhat vague.
- If you want a specific example, think of a case where the number of edges is linear to the number of vertices.
  – For example, if each vertex can only have between 1 and 10 neighbors, than the number of edges can be at most 10*V.

Prim's Algorithm for Sparse Graphs

- If we use an adjacency matrix representation, then we can never do better than $O(V^2)$ time.
- Why?

Prim's Algorithm for Sparse Graphs

- If we use an adjacency matrix representation, then we can never do better than $O(V^2)$ time.
- Why?
  - Because just scanning the adjacency matrix to figure out where the edges are takes $O(V^2)$ time.
  - The adjacency matrix itself has size V*V.

Prim's Algorithm for Sparse Graphs

- If we use an adjacency matrix representation, then we can never do better than $O(V^2)$ time.
- Why?
  - Because just scanning the adjacency matrix to figure out where the edges are takes $O(V^2)$ time.
  - The adjacency matrix itself has size V*V.
- We have already seen an implementation of Prim's algorithm, using adjacency matrices, which achieves $O(V^2)$ running time.
- For sparse graphs, if we want to achieve better running time than $O(V^2)$, we have to switch to an adjacency lists representation.

Prim's Algorithm for Sparse Graphs

- Quick review: what exactly is an adjacency lists representation?

Prim's Algorithm for Sparse Graphs

- Quick review: what exactly is an adjacency lists representation?
  - Each vertex is a number between 0 and V (same as for adjacency matrices).
  - The adjacency information is stored in an array ADJ of lists.
  - ADJ[w] is a list containing all neighbors of vertex w.
- What is the sum of length of all lists in the ADJ array?

Prim's Algorithm for Sparse Graphs

- Quick review: what exactly is an adjacency lists representation?
  - Each vertex is a number between 0 and V (same as for adjacency matrices).
  - The adjacency information is stored in an array ADJ of lists.
  - ADJ[w] is a list containing all neighbors of vertex w.
- What is the sum of length of all lists in the ADJ array?
  - 2*E (each edge is included in two lists).

Prim's Algorithm for Sparse Graphs

- Quick review: what exactly is an adjacency lists representation?
  - Each vertex is a number between 0 and V (same as for adjacency matrices).
  - The adjacency information is stored in an array ADJ of lists.
  - ADJ[w] is a list containing all neighbors of vertex w.
- What is the sum of length of all lists in the ADJ array?
  - 2*E (each edge is included in two lists).

Prim's Algorithm for Sparse Graphs

- For sparse graphs, we will use an implementation of Prim's algorithm based on:
  - A graph representation using **adjacency lists**.
  - A **priority queue** containing the set of edges on the fringe.
- An edge F will be included in this priority queue if: for some vertex w NOT in the tree yet, F is the shortest edge connecting w to vertex in the tree.

Prim's Algorithm - PQ Version

- Initialize a priority queue P.
- v = vertex 0
- While (true)
  - Add v to the spanning tree.
  - Let S = set of edges from v to vertices not yet in the tree to P:
  - If S is empty, exit.
  - For each F = (v, w) in S
    - If another edge F' in P also connects to w, keep the smallest of F and F'.
    - Else insert F to P.
  - F = remove_minimum(P)
  - v = vertex of F not yet in the tree.

Prim's Algorithm - PQ Version

- Running time???

Prim's Algorithm - PQ Version

- Running time? O(E lg V).

Kruskal's Algorithm: Overview

- Prim's algorithm works with a single tree, such that:
  – First, the tree contains a single vertex.
  – The tree keeps growing, until it spans the whole tree.
- Kruskal's algorithm works with a forest (a set of trees).
  – Initially, each tree in this forest is a single vertex.
  – Each vertex in the graph is its own tree.
  – We keep merging trees together, until we end up with a single tree.

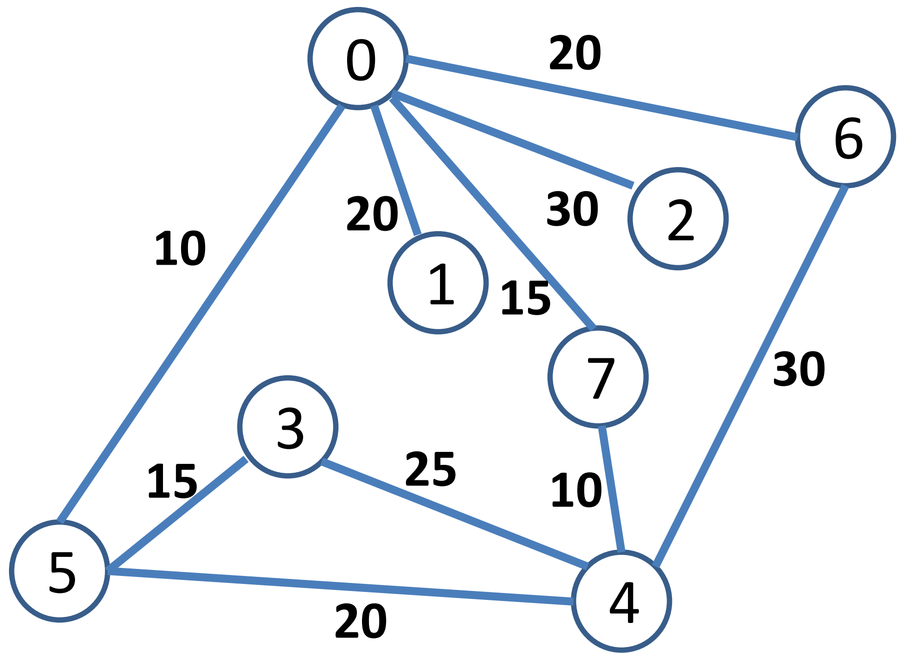
Kruskal's Algorithm: Overview

1. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree.
2. Repeat until the forest contains a single tree:
    3. Find the shortest edge F connecting two trees in the forest.
    4. Connect those two trees into a single tree using edge F.

- As in Prim's algorithm, the abstract description is simple, but we need to think carefully about how exactly to implement these steps.

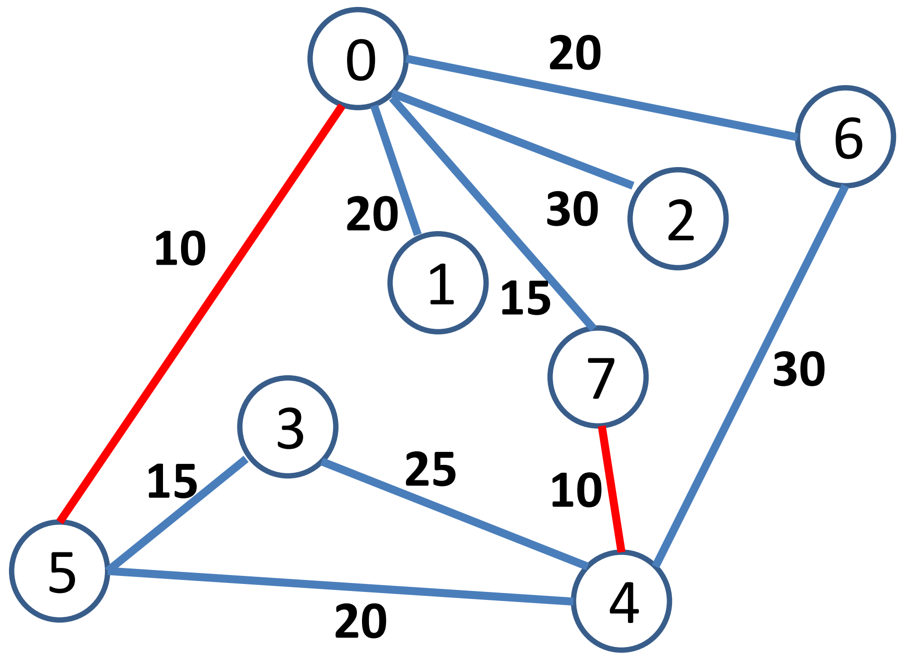
Kruskal's Algorithm: An Example

1. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree.
2. Repeat until the forest contains a single tree:
   3. Find the shortest edge F connecting two trees in the forest.
   4. Connect those two trees into a single tree using edge F.



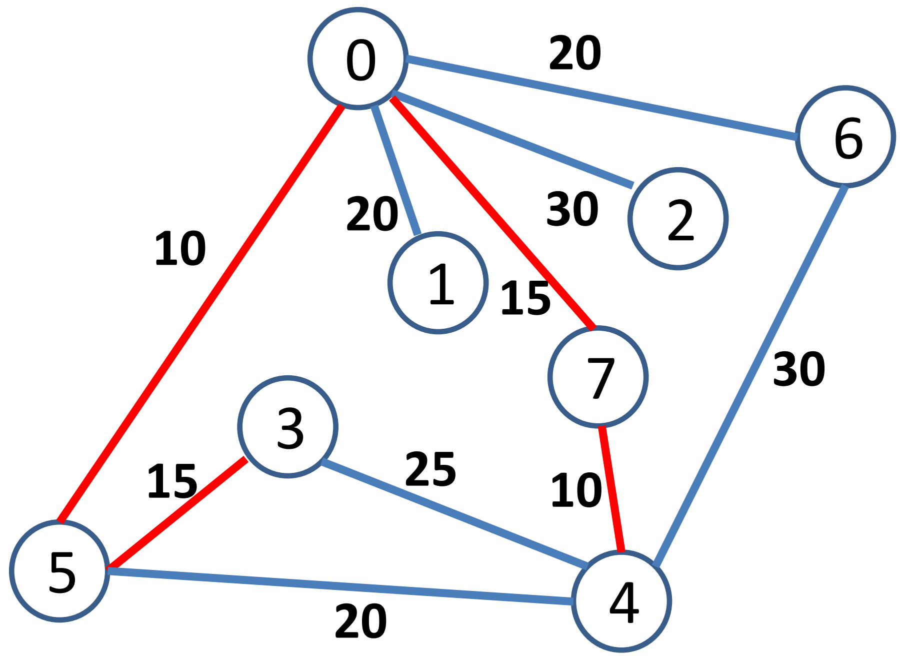
Kruskal's Algorithm: An Example

1. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree.
2. Repeat until the forest contains a single tree:
    3. Find the shortest edge F connecting two trees in the forest.
    4. Connect those two trees into a single tree using edge F.



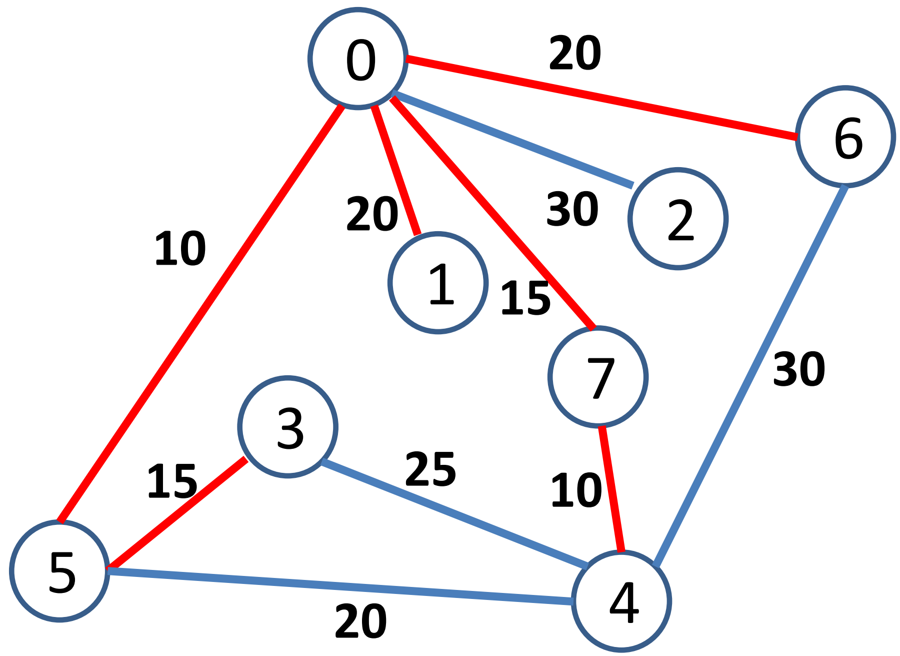
Kruskal's Algorithm: An Example

1. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree.
2. Repeat until the forest contains a single tree:
   3. Find the shortest edge F connecting two trees in the forest.
   4. Connect those two trees into a single tree using edge F.



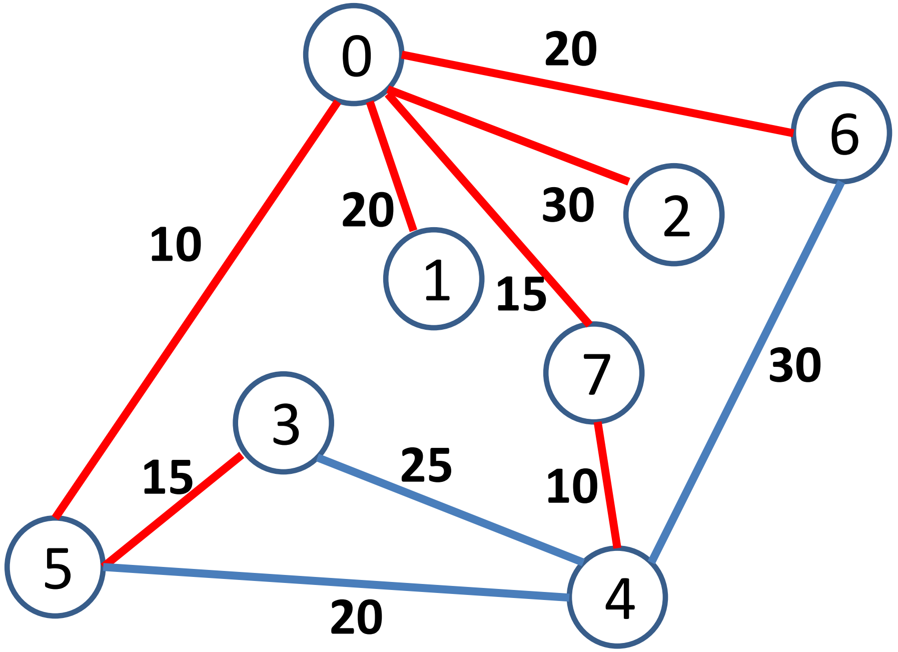
Kruskal's Algorithm: An Example

1. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree.
2. Repeat until the forest contains a single tree:
    3. Find the shortest edge F connecting two trees in the forest.
    4. Connect those two trees into a single tree using edge F.



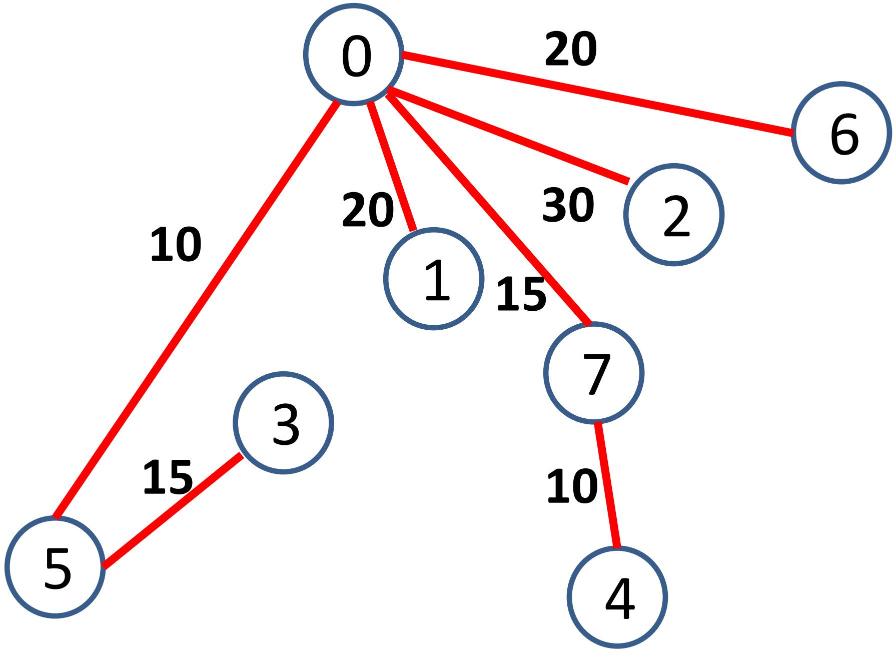
Kruskal's Algorithm: An Example

1. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree.
2. Repeat until the forest contains a single tree:
   3. Find the shortest edge F connecting two trees in the forest.
   4. Connect those two trees into a single tree using edge F.



Kruskal's Algorithm: An Example

1. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree.
2. Repeat until the forest contains a single tree:
    3. Find the shortest edge F connecting two trees in the forest.
    4. Connect those two trees into a single tree using edge F.



Kruskal's Algorithm: Simple Implementation

Assume graphs are represented usind **adjacency lists**.

1. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree.
   - How? We will use the same representation for forests that we used for union-find.
   - We will have an id array, where each vertex will point to its parent.
   - The root of each tree will be the ID for that tree.

- Time it takes for this step ???

Kruskal's Algorithm: Simple Implementation

Assume graphs are represented usind **adjacency lists**.

1. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree.
   - How? We will use the same representation for forests that we used for union-find.
   - We will have an id array, where each vertex will point to its parent.
   - The root of each tree will be the ID for that tree.

- Time it takes for this step? O(V)

Kruskal's Algorithm: Simple Implementation

2. Repeat until the forest contains a single tree:
   3. Find the shortest edge F connecting two trees in the forest.
   - Initialize F to some edge with infinite weight.
   - For each edge F' connecting (v, w):
     - Determine if v and w belong to the same tree in the forest.
     - If so, update F to be the shortest of F and F'.

   4. Connect those two trees into a single tree using edge F.

Kruskal's Algorithm: Simple Implementation

2. Repeat until the forest contains a single tree:
   3. Find the shortest edge F connecting two trees in the forest.
   - Initialize F to some edge with infinite weight.
   - For each edge F' connecting (v, w):
     - Determine if v and w belong to the same tree in the forest. **HOW?**
     - If so, update F to be the shortest of F and F'.

   4. Connect those two trees into a single tree using edge F. **HOW?**

Kruskal's Algorithm: Simple Implementation

2. Repeat until the forest contains a single tree:
   3. Find the shortest edge F connecting two trees in the forest.
   - Initialize F to some edge with infinite weight.
   - For each edge F' connecting (v, w):
     - Determine if v and w belong to the same tree in the forest. **HOW? By comparing find(v) with find(w). Time:**
     - If so, update F to be the shortest of F and F'.

   4. Connect those two trees into a single tree using edge F. **HOW? By calling union(v, w). Time:**

Kruskal's Algorithm: Simple Implementation

2. Repeat until the forest contains a single tree:
   3. Find the shortest edge F connecting two trees in the forest.
   - Initialize F to some edge with infinite weight.
   - For each edge F' connecting (v, w):
     - Determine if v and w belong to the same tree in the forest. **HOW? By comparing find(v) with find(w). Time: O(lg V)**
     - If so, update F to be the shortest of F and F'.

   4. Connect those two trees into a single tree using edge F. **HOW? By calling union(v, w). Time: O(1)**

Kruskal's Algorithm: Simple Implementation

2. Repeat until the forest contains a single tree:
   Total time for all iterations:
   3. Find the shortest edge F connecting two trees in the forest. **Time:**
   – Initialize F to some edge with infinite weight.
   – For each edge F' connecting (v, w):
     - Determine if v and w belong to the same tree in the forest. **HOW? By comparing find(v) with find(w). Time: O(lg V)**
     - If so, update F to be the shortest of F and F'.

   4. Connect those two trees into a single tree using edge F. **HOW? By calling union(v, w). Time: O(1)**

Kruskal's Algorithm: Simple Implementation

2. Repeat until the forest contains a single tree:
   Total time for all iterations: O(V*E*lg(V))
   3. Find the shortest edge F connecting two trees in the forest. **Time: O(E*lg(V))**
   – Initialize F to some edge with infinite weight.
   – For each edge F' connecting (v, w):
     - Determine if v and w belong to the same tree in the forest. **HOW? By comparing find(v) with find(w). Time: O(lg V)**
     - If so, update F to be the shortest of F and F'.

   4. Connect those two trees into a single tree using edge F. **HOW? By calling union(v, w). Time: O(1)**

Running Time for Simple Implementation

1. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree.
2. Repeat until the forest contains a single tree:
   3. Find the shortest edge F connecting two trees in the forest.
   4. Connect those two trees into a single tree using edge F.

- Running time for simple implementation: O(V*E*lg(V)).

Kruskal's Algorithm: Faster Version

1. Sort all edges, save result in array K.
2. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree.
3. For each edge F in K (in ascending order).
    4. If F is connecting two trees in the forest:
        5. Connect the two trees with F.
        6. If the forest is left with a single tree, break (we are done).

Kruskal's Algorithm: Faster Version

1. Sort all edges, save result in array K. **Time?**
2. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree. **Time?**
3. For each edge in K (in ascending order). **Time?**
    4. If F is connecting two trees in the forest: **Time?**
        5. Connect the two trees with F. **Time?**
        6. If the forest is left with a single tree, break (we are done).

Kruskal's Algorithm: Faster Version

1. Sort all edges, save result in array K. **Time: O(E lg E)**
2. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree. **Time: O(V)**
3. For each edge in K (in ascending order). **Time: O(E lg V)**
    4. If F is connecting two trees in the forest:
       Time? O(lg V), two find operations
        5. Connect the two trees with F. **Time: O(1), union operation**
        6. If the forest is left with a single tree, break (we are done).

- Overall running time???

Kruskal's Algorithm: Faster Version

1. Sort all edges, save result in array K. **Time: O(E lg E)**
2. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree. **Time: O(V)**
3. For each edge in K (in ascending order). **Time: O(E lg V)**
    4. If F is connecting two trees in the forest:
       Time? O(lg V), two find operations
        5. Connect the two trees with F. **Time: O(1), union operation**
        6. If the forest is left with a single tree, break (we are done).

- Overall running time: O(E lg E).

Kruskal's Algorithm - PQ Version

- In the previous implementation, we sort edges at the beginning.
  - This takes O(E lg E) time, which dominates the running time of the algorithm.
  - Thus, the entire algorithm takes O(E lg E) time.
- We can do better if, instead of sorting all edges at the beginning, we instead insert all edges into a priority queue.
  - How long does that take, if we use a heap?

Kruskal's Algorithm - PQ Version

- In the previous implementation, we sort edges at the beginning.
  - This takes O(E lg E) time, which dominates the running time of the algorithm.
  - Thus, the entire algorithm takes O(E lg E) time.
- We can do better if, instead of sorting all edges at the beginning, we instead insert all edges into a priority queue.
  - How long does that take, if we use a heap?
  - O(E) time.
- We can also do better if, for the find operation, we use the most efficient version discussed in the textbook.
  - That version flattens paths that it traverses.
  - Running time: O(lg* V).
  - lg*(V) is the number of times we need to apply lg to V to obtain 1.

Detour: lg*

- lg*(2) = ?
- lg*(4) = ?
- lg*(16) = ?

Detour: lg*

- lg*(2) = 1, because lg(2) = 1.
- lg*(4) = 2, because lg(lg(4)) = 1.
- lg*(16) = 3, because lg(lg(lg(16))) = 1.
- lg*(???) = 4
- lg*(???) = 5

Detour: lg*

- lg*(2) = 1, because lg(2) = 1.
- lg*(4) = 2, because lg(lg(4)) = 1.
- lg*(16) = 3, because lg(lg(lg(16))) = 1.
- lg*(65536) = 4, because lg(65536) = 16.
- lg*(2^{65536}) = 5, because lg(2^{65536}) = 65536.

- I don't expect we will get to deal with data sizes larger than 2^{65536} in our lifetime.
- Thus, lg* effectively has 5 as an upper bound, so for practical purposes we can treat it as a constant.

Kruskal's Algorithm - PQ Version

1. Initialize a heap with the edges (using weight as key).

2. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree.
3. While (true).
    4. F = remove_mininum(heap).
    5. If F is connecting two trees in the forest:

        6. Connect the two trees with F.
        7. If the forest is left with a single tree, break (we are done).

Kruskal's Algorithm - PQ Version

1. Initialize a heap with the edges (using weight as key). **Time?**
2. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree. **Time?**
3. While (true). **Time?**
    4. F = remove_mininum(heap). **Time?**
    5. If F is connecting two trees in the forest: **Time?**
        6. Connect the two trees with F. **Time?**
        7. If the forest is left with a single tree, break (we are done).

- **Overall running time?**

Kruskal's Algorithm - PQ Version

1. Initialize a heap with the edges (using weight as key). **Time? O(E)**
2. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree. **Time? O(V)**
3. While (true). **Time? X lg V.** **X: number of iterations.**
    4. F = remove_mininum(heap). **Time? O(lg E)**
    5. If F is connecting two trees in the forest: **Time? O(lg* V), find operation**
        6. Connect the two trees with F. **Time? O(1)**
        7. If the forest is left with a single tree, break (we are done).

- **Overall running time? E + X lg V.**

Kruskal's Algorithm - PQ Version

1. Initialize a heap with the edges (using weight as key).
2. Initialize a forest (a collection of trees), by defining each vertex to be its own separate tree.
3. While (true). **Time? X lg V.** **X: number of iterations.**
    4. F = remove_mininum(heap). **Time? O(lg E)**
    5. If F is connecting two trees in the forest:
        6. Connect the two trees with F. **Time? O(1)**
        7. If the forest is left with a single tree, break (we are done).

- Overall running time? E + X lg V.
  - X is the number of edges in the graph with weight <= the maximum weight of an edge in the final MST.
  - $E < V^2$, so $\lg E < 2 \lg V$, so $O(\lg E) = O(\lg V)$.