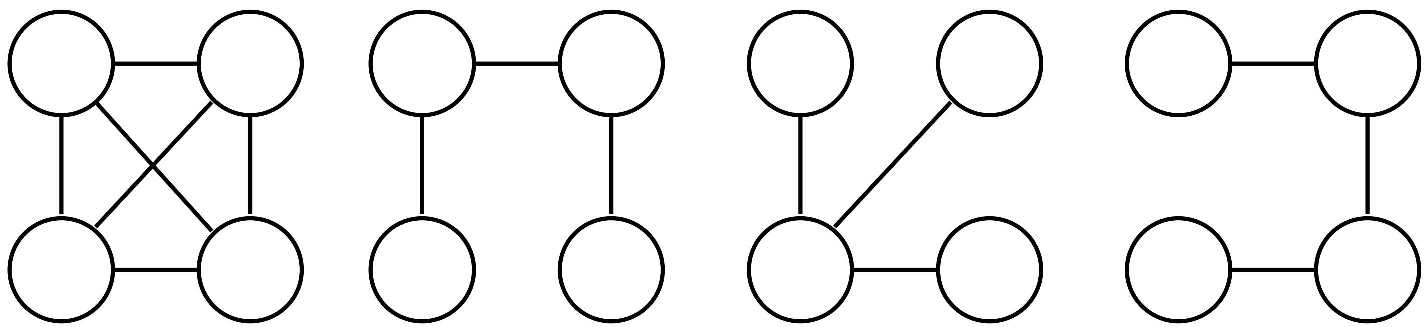
Minimum Spanning Trees

Prof. Ki-Hoon Lee
Dept. of Computer Engineering
Kwangwoon University

Spanning Trees

- A spanning tree is a minimal subgraph, G', of G such that V(G') = V(G), and G' is connected



A complete graph and three of its spanning trees

Spanning Trees (cont.)

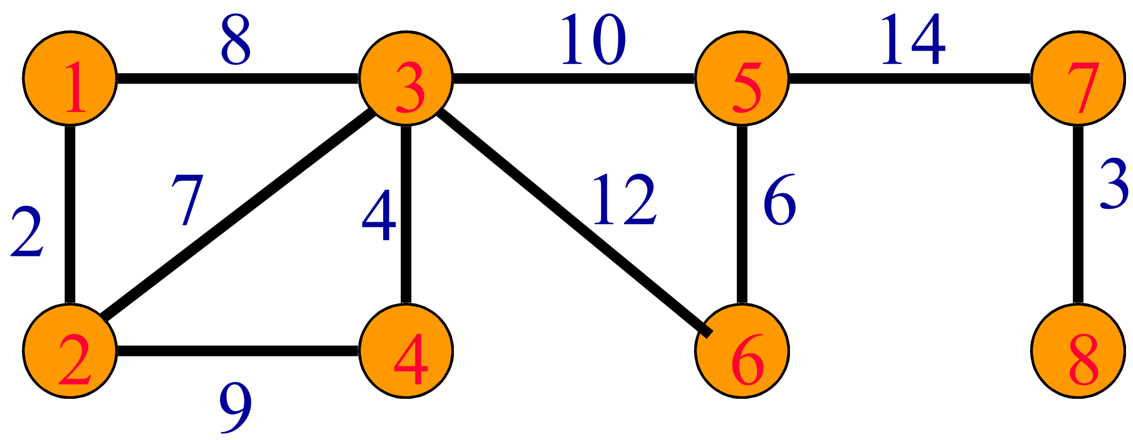
- Any connected graph with n vertices must have at least (n – 1) edges
- All connected graphs with n – 1 edges are trees

→ A spanning tree for a graph with n vertices has n – 1 edges

Minimum Spanning Tree (MST)

- The cost of a spanning tree of a weighted, undirected graph = the sum of the costs (weights) of the edges in the spanning tree
- A *minimum spanning tree* (*MST*) is a spanning tree of least cost
- Application: communication network design

Example



- Network has n = 8 vertices.
- Spanning tree has n – 1 = 7 edges.
- Find a MST

Greedy Methods

- We construct a solution in stages
- At each stage, we make the best decision (using some criterion) possible at the time
  - Typically, the decision is based on either a least-cost or a highest profit criterion
- In many problems, a greedy method may yield *locally optimal* solutions that approximate a *global optimal* solution in a reasonable time

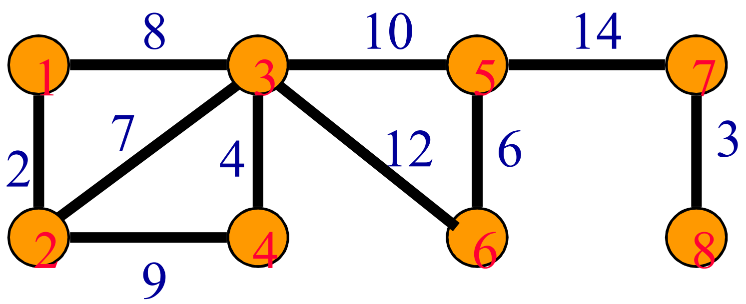
Greedy Methods for MSTs

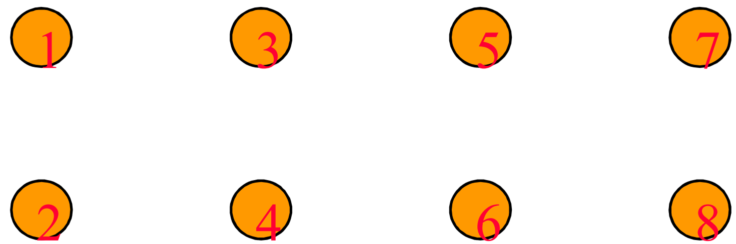
- To construct a MST, we use a least-cost criterion
- Our solution must satisfy the following constraints:
  1. We must use only edges within the graph
  2. We must use exactly $n - 1$ edges
  3. We may not use edges that produce a cycle
- Three methods: Kruskal's, Prim's, and Sollin's methods

Kruskal's Method

- Build a minimum-cost spanning tree T by adding edges to T one at a time
- Select the edges for inclusion in T in non-decreasing order of their cost
- An edge is added to T if it does not form a cycle with the edges that are already in T

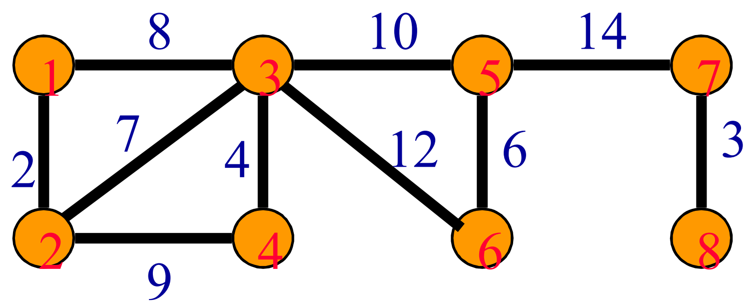
Kruskal's Method (cont.)

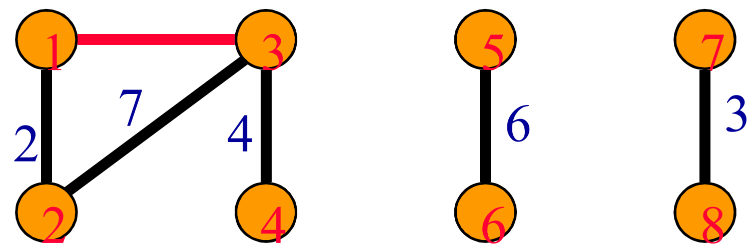




- Start with a forest that has no edges.
- Consider edges in ascending order of cost.
- Edge (1,2) is considered first and added to the forest.

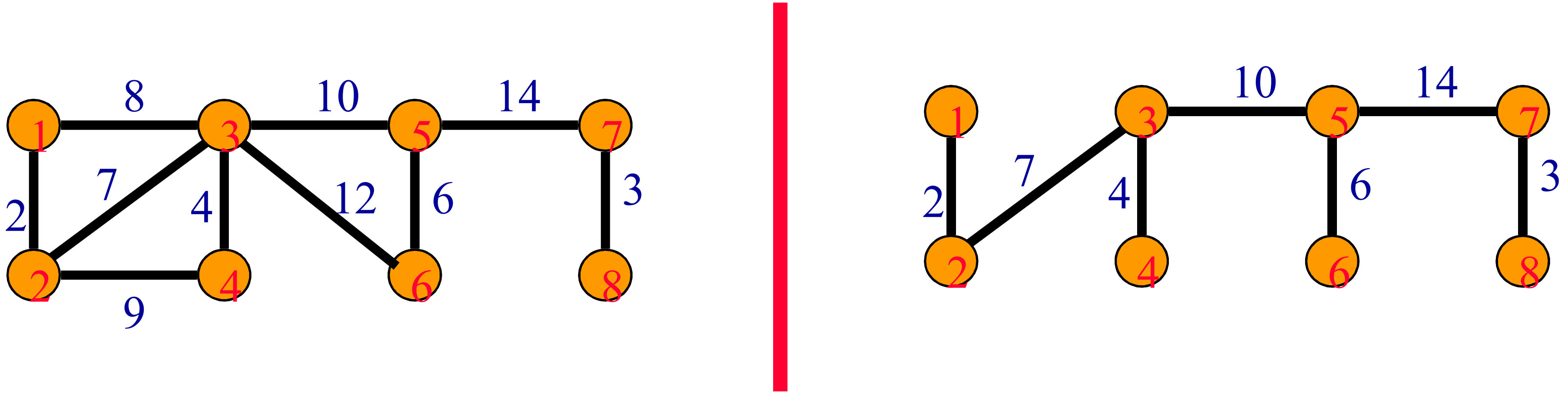
Kruskal’s Method (cont.)





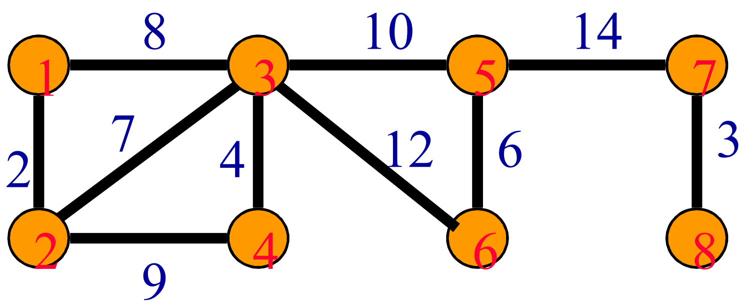
- Edge (7,8) is considered next and added.
- Edge (3,4) is considered next and added.
- Edge (5,6) is considered next and added.
- Edge (2,3) is considered next and added.
- Edge (1,3) is considered next and rejected because it creates a cycle.

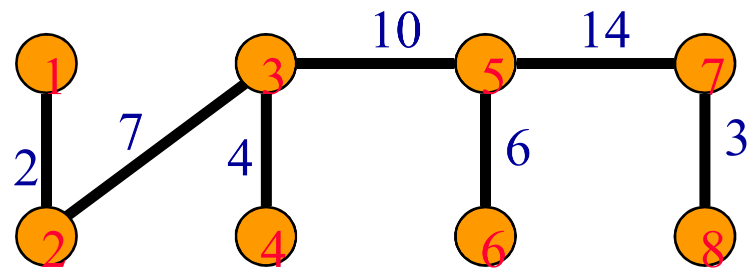
Kruskal's Method (cont.)



- Edge (2,4) is considered next and rejected because it creates a cycle.
- Edge (3,5) is considered next and added.
- Edge (3,6) is considered next and rejected.
- Edge (5,7) is considered next and added.

Kruskal's Method (cont.)





- n – 1 edges have been selected and no cycle formed.
- So we must have a spanning tree.
- Cost is 46.
- MST is unique when all edge costs are different.

Kruskal's Method (cont.)

```
T = ∅;
while ((T contains less than n–1 edges) &&
          (E not empty)) {
   choose an edge (v, w) from E of lowest cost;
   delete (v, w) from E;
   if ((v, w) does not create a cycle in T)
      add (v, w) to T;
   else discard (v, w);
}
if (T contains fewer than n–1 edges)
   cout << "no spanning tree" << endl;
```

Implementation

- choose an edge *(v, w)* from *E* of lowest cost;
  - Use a min heap (O(log e))

- if ((*v*, *w*) does not create a cycle in *T*) add (*v*, *w*) to *T*;
  - Determine if the vertices *v* and *w* are already connected by the earlier selection of edges. If they are not, then (*v*, *w*) is to be added to *T*.
  - To determine this, place all vertices in the same connected component of *T* into a set. Then, *v* and *w* are connected in *T* iff they are in the same set.
  - Use the methods *WeightedUnion* and *CollapsingFind* for disjoint sets

Data Structures for Kruskal’s Method

Edge set E.

Operations are:

- Is E empty?
- Select and remove a least-cost edge.

Use a min heap of edges.

- Initialize. O(e) time.
- Remove and return least-cost edge. O(log e) time.

Data Structures for Kruskal's Method

Set of selected edges T.

Operations are:

- Does T have n - 1 edges?
- Does the addition of an edge (u, v) to T result in a cycle?
- Add an edge to T.

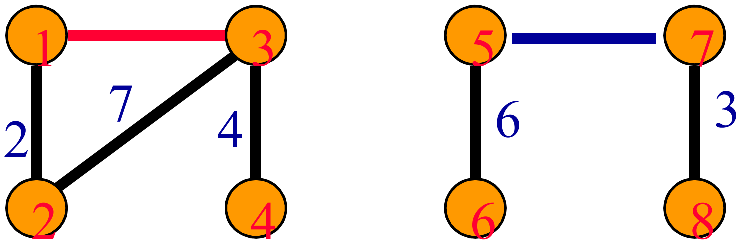
Data Structures for Kruskal's Method

Use an array for the edges of T.

- Does T have n - 1 edges?
  - Check number of edges in array. O(1) time.
- Does the addition of an edge (u, v) to T result in a cycle?
  - Not easy.
- Add an edge to T.
  - Add at right end of edges in array. O(1) time.

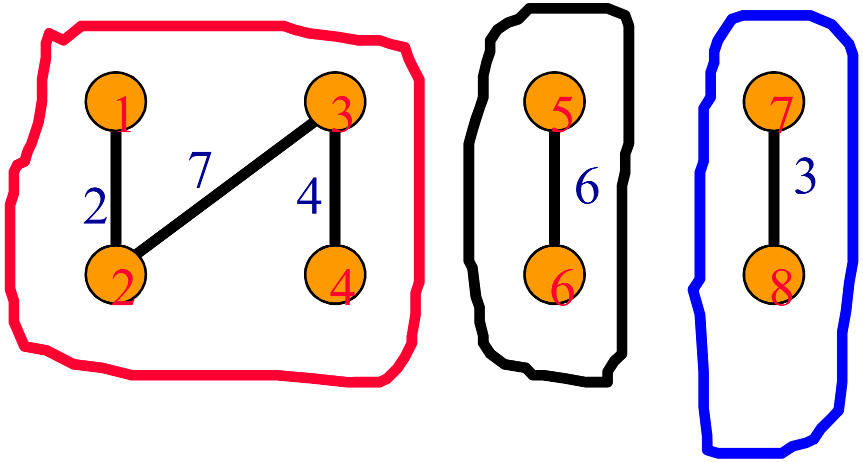
Data Structures for Kruskal's Method

Does the addition of an edge (u, v) to T result in a cycle?



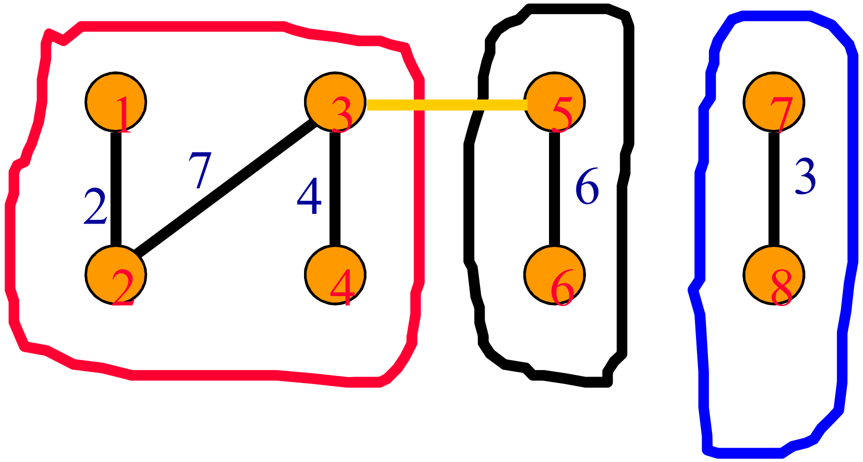
- Each component of T is a tree.
- When u and v are in the same component, the addition of the edge (u,v) creates a cycle.
- When u and v are in the different components, the addition of the edge (u,v) does not create a cycle.

Data Structures for Kruskal's Method



- Each component of T is defined by the vertices in the component.
- Represent each component as a set of vertices.
  - {1, 2, 3, 4}, {5, 6}, {7, 8}
- Two vertices are in the same component iff they are in the same set of vertices.

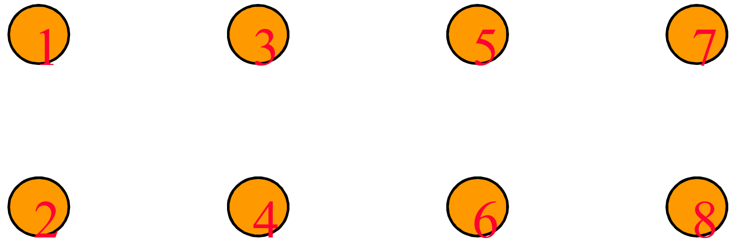
Data Structures for Kruskal's Method



- When an edge (u, v) is added to T, the two components that have vertices u and v combine to become a single component.
- In our set representation of components, the set that has vertex u and the set that has vertex v are united.
  - {1, 2, 3, 4} U {5, 6} => {1, 2, 3, 4, 5, 6}

Data Structures for Kruskal's Method

- Initially, T is empty.



- Initial sets are:
  - {1} {2} {3} {4} {5} {6} {7} {8}
- Does the addition of an edge (u, v) to T result in a cycle? If not, add edge to T.

s1 = Find(u); s2 = Find(v);

if (s1 != s2) Union(s1, s2);

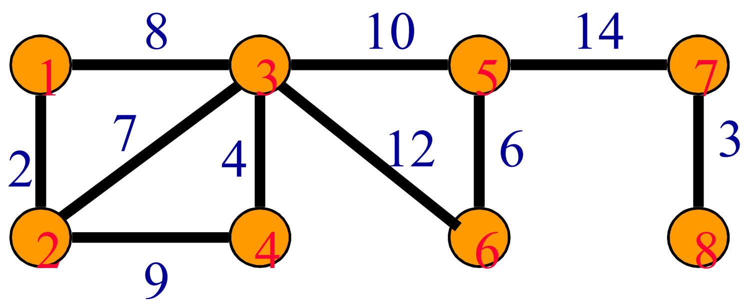
Data Structures for Kruskal's Method

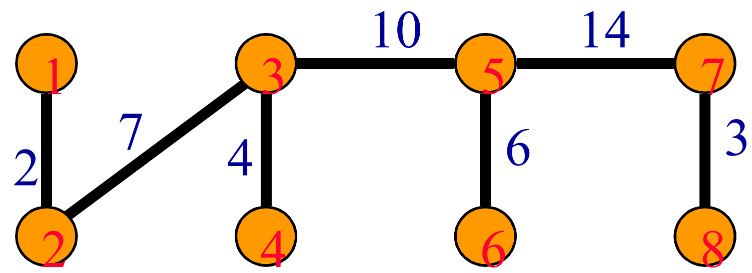
- Use fast solution for disjoint sets.
- Initialize.
  - $O(n)$ time.
- At most $2e$ finds and $n-1$ unions.
  - Very close to $O(n + e)$.
- Min heap operations to get edges in increasing order of cost take $O(e \log e)$.
- Overall complexity of Kruskal's method is $O(n + e \log e)$.

Prim's Method

- At each stage of the algorithm, the set of selected edges forms a tree
- Begins with a tree T that contains a single vertex
- Add a least-cost edge (u, v) to T such that $T \cup \{(u, v)\}$ is also a tree, where exactly one of u or v is in T

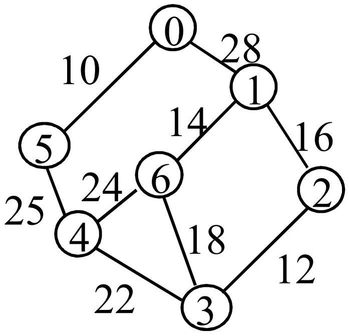
Prim's Method (cont.)

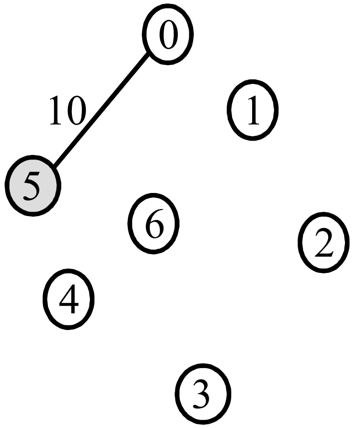




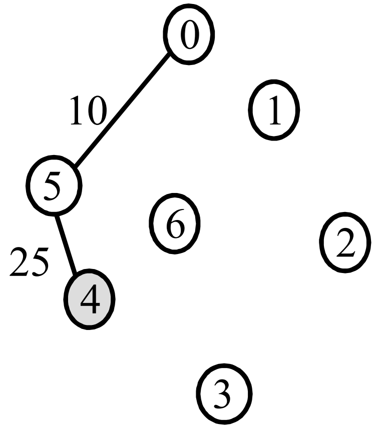
- Start with any single vertex tree.
- Get a 2-vertex tree by adding a cheapest edge.
- Get a 3-vertex tree by adding a cheapest edge.
- Grow the tree one edge at a time until the tree has n - 1 edges (and hence has all n vertices).

Prim's Method (cont.)

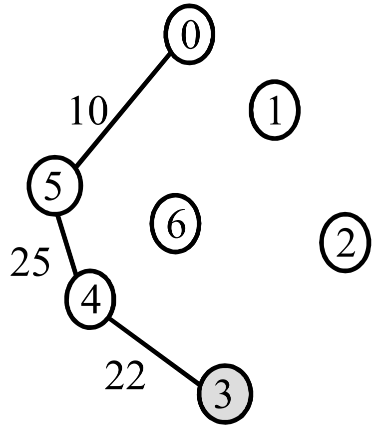




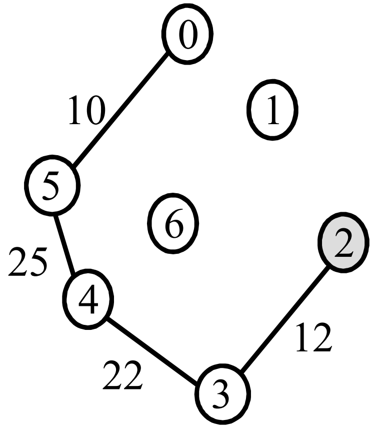
(a)



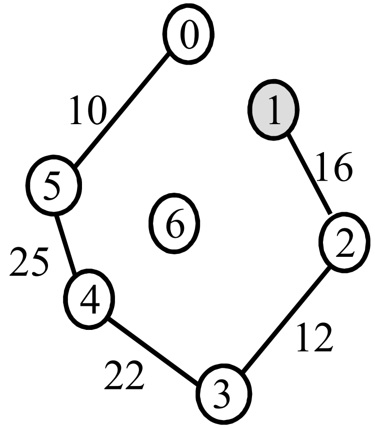
(b)



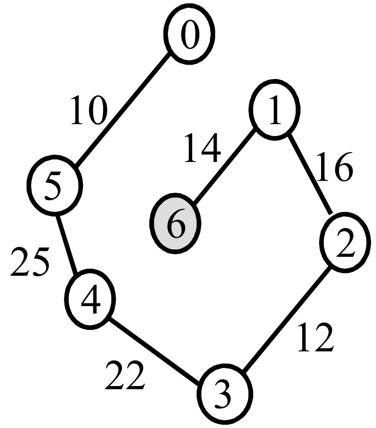
(c)



(d)



(e)



(f)

Prim's Method (cont.)

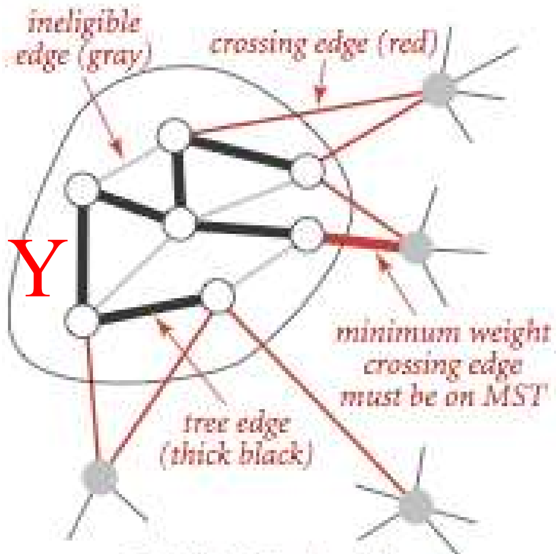
```
// Assume that G has at least one vertex.
Y = {0}; // start with vertex 0 and no edges
for (T = ∅; T contains fewer than n – 1 edges;  add (u, v) to T)
{
   Let (u, v) be a least-cost edge such that u ∈ Y and v ∉ Y;
   if (there is no such edge) break;
   add v to Y;
}
if (T contains fewer than n – 1 edges)
   cout << "no spanning tree" << endl;
```

Prim's Method (cont.)

- We associate with each vertex v not in Y a vertex nearest(v) such that
  - nearest(v) ∈ Y and
  - cost(nearest(v), v) is the minimum over all such choices for nearest(v)

  (We assume that cost(v, w) = ∞ if (v, w) ∉ E)

- The next edge to add to T is such that cost(nearest(v), v) is the minimum and v ∉ T



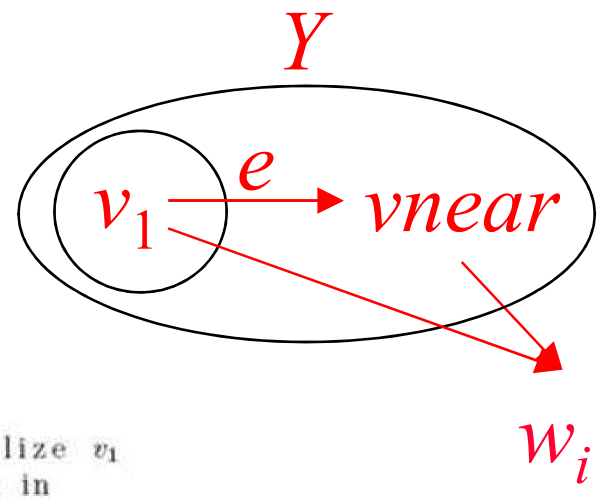
Prim's MST algorithm

Prim's Method (cont.)

- Prim's algorithm starts with an empty subset of edges F and a subset of vertices Y initialized to contain an arbitrary vertex.
- We will initialize Y to $\{v_1\}$. A vertex nearest to Y is a vertex in $V - Y$ that is connected to a vertex in Y by an edge of minimum weight.
- Assume that v_2 is nearest to Y when $Y = \{v_1\}$. The vertex that is nearest to Y is added to Y and the edge is added to F. Ties are broken arbitrarily.
- In this case, v_2 is added to Y, and (v_1, v_2) is added to F. This process of adding nearest vertices is repeated until $Y = V$.

```
void prim (int n,
           const number W[][],
           set_of_edges& F)
{
  index i, vnear;
  number min;
  edge e;
  index nearest[2..n];
  number distance[2..n];

  F = ∅;
  for (i = 2; i <= n; i++){
     nearest[i] = 1;              // For all vertices, initialize v1
     distance[i] = W[1][i];       // to be the nearest vertex in
  }                               // Y and initialize the distance
                                  // from Y to be the weight
                                  // on the edge to v1.
  repeat (n - 1 times){
                                       // Add all n - 1 vertices to Y.
     min = ∞;
     for (i = 2; i <= n; i++)          // Check each vertex for
        if (0 ≤ distance[i] < min){    // being nearest to Y.
           min = distance[i];
           vnear = i;
        }
     e = edge connecting vertices indexed
         by vnear and nearest[vnear];
     add e to F;
     distance[vnear] = -1;             // Add vertex indexed by
     for (i = 2; i <= n; i++)          // vnear to Y.
        if (W[i][vnear] < distance[i]){ // For each vertex not in
           distance[i] = W[i][vnear];  // Y, update its distance
           nearest[i] = vnear;         // from Y.
        }
  }
}
```



Minimum Spanning Tree Methods

- Can prove that all the three methods result in a minimum-cost spanning tree.
- Prim's method is fastest.
  - $O(n^2)$ using an implementation similar to that of Dijkstra's shortest-path method.
  - $O(e + n \log n)$ using a Fibonacci heap.
- Kruskal's uses union-find trees to run in $O(n + e \log e)$ time.