CS 4102: Greedy Algorithms

- Topics covered for greedy algorithms
  - General principles
  - Making change
  - Knapsack problems
  - Activity Selection
  - Minimum spanning trees: Prim's, Kruskal's algorithms
  - Single-source shortest path: Dijkstra's algorithm
  - Approximation algorithms

Greedy Method: Overview

- Optimization problems: terminology
  - Solutions judged on some criteria:
    Objective function

  Example: Sum of edge weights in path is smallest
  - A solution must meet certain constraints
    A solution is *feasible*

  Example: All edges in solution are in graph, form a simple path
  - One (or more) feasible solutions that scores highest (by the objective function) is the *optimal solution(s)*

Greedy Method: Overview

- Greedy strategy:
  - Build solution by stages, adding one item to partial solution found so far
  - At each stage, make locally optimal choice based on the *greedy rule* (sometimes called the *selection function)*
    - Locally optimal, I.e. best given what info we have now
  - Irrevocable, a choice can't be un-done
  - Sequence of locally optimal choices leads to globally optimal solution (hopefully)
    - Must prove this for a given problem!
    - Approximation algorithms, heuristics

Making Change

- Remember? We did this one in class on Day 1
- Inputs:
  - Value N of the change to be returned
  - An unlimited number of coins of values d1, d2,.., dk
- Output: the smallest possible set of coins that sums to N
- Objective function? Smallest set
- Constraints on feasible solutions? Must sum to N
- Greedy rule: choose coin of largest value that is less than N - Sum(coins chosen so far)
- Always optimal? Depends on set of coin values

Algorithm 7.1.1 Greedy Coin Changing

This algorithm makes change for an amount A using coins of denominations

$$denom[1] > denom[2] > \cdots > denom[n] = 1.$$

```
Input Parameters: denom,A
Output Parameters: None
greedy_coin_change(denom,A) {
   i = 1
   while (A > 0) {
      c = A/denom[i]
      println("use " + c + " coins of denomination " +
                                 denom[i])
      A = A - c * denom[i]
      i = i + 1
   }
}
```

Knapsack Problems

- Section 7.6 in text
- Inputs:
  - n items, each with a weight w_i and a value v_i
  - capacity of the knapsack, C
- Output:
  - Fractions for each of the n items, x_I
  - Chosen to maximize total profit but not to exceed knapsack capacity

Two Types of Knapsack Problem

- 0/1 knapsack problem
  - Each item is discrete. Must choose all of it or none of it. So each x_i is 0 or 1
  - Greedy approach does not produce optimal solutions
  - But another approach, dynamic programming, does
- Continuous knapsack problem
  - Can pick up fractions of each item
  - The correct selection function yields a greedy algorithm that produces optimal results

Greedy Rule for Knapsack?

- Build up a partial solution by choosing x_i for one item until knapsack is full (or no more items). Which item to choose?
- There are several choices. Pick one and try on this:
  - n = 3, C = 20
  - weights = (18, 15, 10)
  - values = (25, 24, 15)
- What answer do you get?
- The optimal answer is: (0, 1, 0.5), total=31.5
  Can you verify this?

Possible Greedy Rules for Knapsack

- Build up a partial solution by choosing x_i for one item until knapsack is full (or no more items). Which item to choose?
  - Maybe this: take as much as possible of the remaining item that has largest value, v_i
  - Or maybe this: take as much as possible of the remaining items that has smallest weight, w_i
  - Neither of these produce optimal values! The one that does "combines" these two approaches.
    - Use ratio of profit-to-weight

Example Knapsack Problem

- For this example:
  - n = 3, C = 20
  - weights = (18, 15, 10)
  - values = (25, 24, 15)
- Ratios = (25/18, 24/15, 15/10)
  = (1.39, 1.6, 1.5)

- The optimal answer is: (0, 1, 0.5)

Activity-Selection Problem

- Problem: You and your classmates go on Semester at Sea
  - Many exciting activities each morning
  - Each starting and ending at different times
  - Maximize your "education" by doing as many as possible. (They're all equally good!)
- Welcome to the *activity selection problem*

The Activities!

Id	Start	End	Activity
1	9:00	10:45	Fractals, Recursion and Crayolas
2	9:15	10:15	Tropical Drink Engineering with Prof. Bloomfield
3	9:30	12:30	Managing Keyboard Fatigue with Swedish Massage
4	9:45	10:30	Applied ChemE: Suntan Oil or Lotion?
5	9:45	11:15	Optimization, Greedy Algorithms, and the Buffet Line
6	10:15	11:00	Hydrodynamics and Surfing
7	10:15	11:30	Computational Genetics and Infectious Diseases
8	10:30	11:45	Turing Award Speech Karaoke
9	11:00	12:00	Pool Tanning for Pale Engineers
10	11:00	12:15	Mechanics, Dynamics and Shuffleboard Physics
11	12:00	12:45	Discrete Math Applications in Gambling

Generalizing Start, End

Id	Start	End	Len	Activity
1	0	6	7	Fractals, Recursion and Crayolas
2	1	4	4	Tropical Drink Engineering with Prof. Bloomfield
3	2	13	12	Managing Keyboard Fatigue with Swedish Massage
4	3	5	3	Applied ChemE: Suntan Oil or Lotion?
5	3	8	6	Optimization, Greedy Algorithms, and the Buffet Line
6	5	7	3	Hydrodynamics and Surfing
7	5	9	5	Computational Genetics and Infectious Diseases
8	6	10	5	Turing Award Speech Karaoke
9	8	11	4	Pool Tanning for Pale Engineers
10	8	12	5	Mechanics, Dynamics and Shuffleboard Physics
11	12	14	3	Discrete Math Applications in Gambling

Greedy Approach

1. Select a first item.
2. Eliminate items that are incompatible with that item. (I.e. they overlap.)
3. Apply the *greedy rule* (AKA *selection function*) to pick the next item.
4. Go to Step 2

What is a good greedy rule for selecting next item?

Some Possibilities

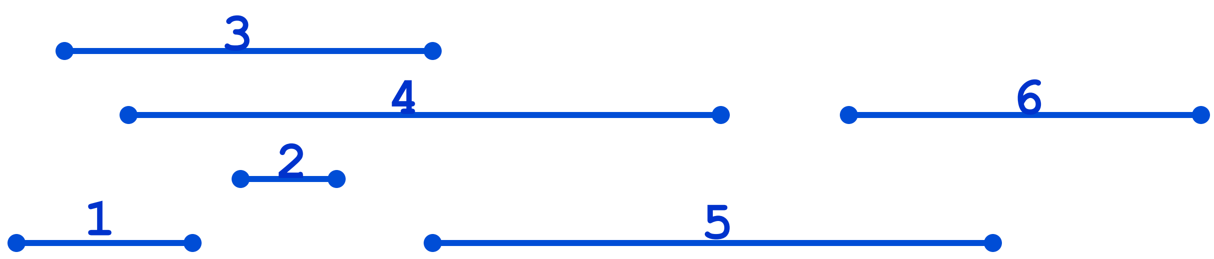
- Pick the next compatible one that starts earliest
- Pick the shortest one
- Pick the one that has the least conflicts (i.e. overlaps)

Activity-Selection

- Formally:
  - Given a set *S* of *n* activities

    s_i = start time of activity *i*

    f_i = finish time of activity *i*
  - Find max-size subset *A* of compatible activities



- **Assume (wlog) that $f_1 \le f_2 \le \ldots \le f_n$**

Activity Selection: Optimal Substructure

- Let k be the minimum activity in A (i.e., the one with the earliest finish time). Then A - $\{k\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_k\}$
  - In words: once activity #1 is selected, the problem reduces to finding an optimal solution for activity-selection over activities in S **compatible** with #1
  - Proof: if we could find optimal solution B' to S' with $|B| > |A - \{k\}|$,
    - Then $B \cup \{k\}$ is compatible
    - And $|B \cup \{k\}| > |A|$

Activity Selection: A Greedy Algorithm

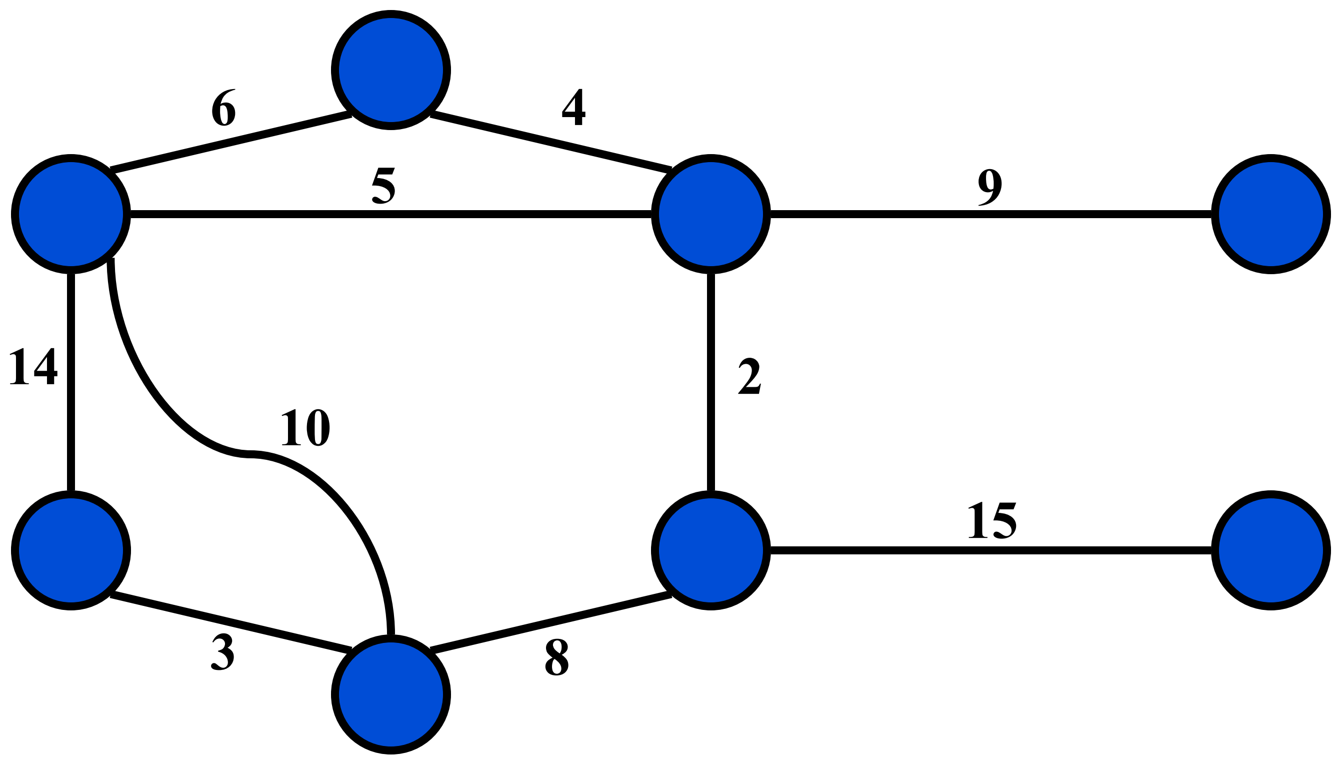
- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Intuition is even more simple:
  - Always pick next activity that finishes earliest

Back to Semester at Sea…

Id	Start	End	Len	Activity
2	1	4	4	Tropical Drink Engineering with Prof. Bloomfield
4	3	5	3	Applied ChemE: Suntan Oil or Lotion?
1	0	6	7	Fractals, Recursion and Crayolas
6	5	7	3	Hydrodynamics and Surfing
5	3	8	6	Optimization, Greedy Algorithms, and the Buffet Line
7	5	9	5	Computational Genetics and Infectious Diseases
8	6	10	5	Turing Award Speech Karaoke
9	8	11	4	Pool Tanning for Pale Engineers
10	8	12	5	Mechanics, Dynamics and Shuffleboard Physics
3	2	13	12	Managing Keyboard Fatigue with Swedish Massage
11	12	14	3	Discrete Math Applications in Gambling

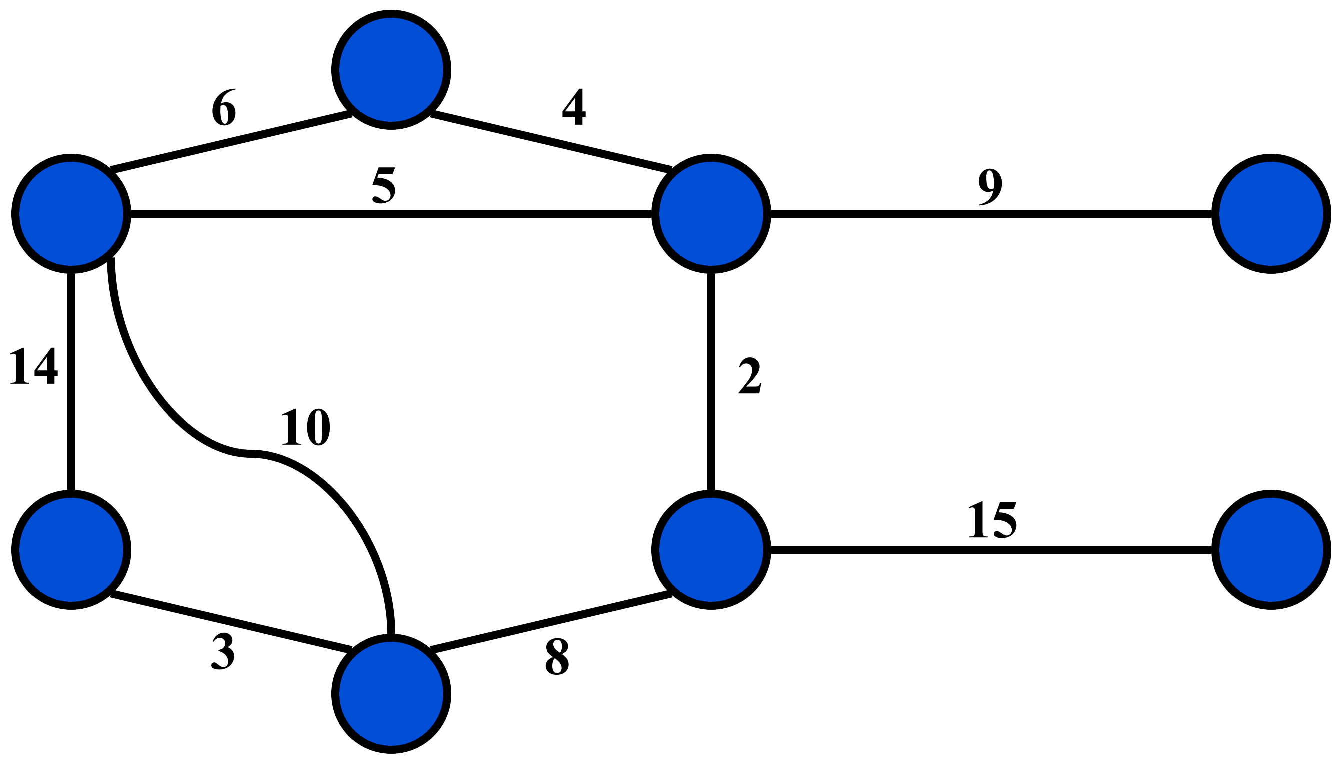
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph:



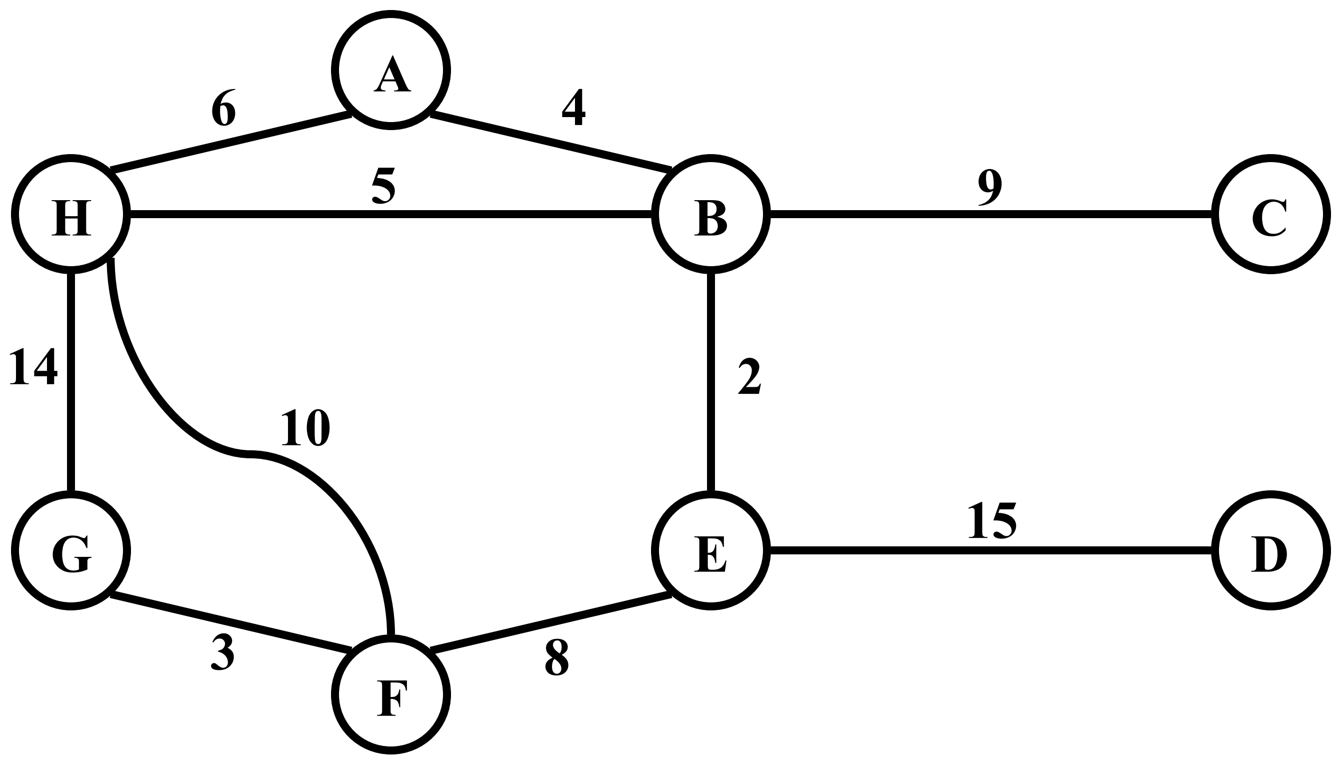
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph, find a *spanning tree* using edges that minimize the total weight



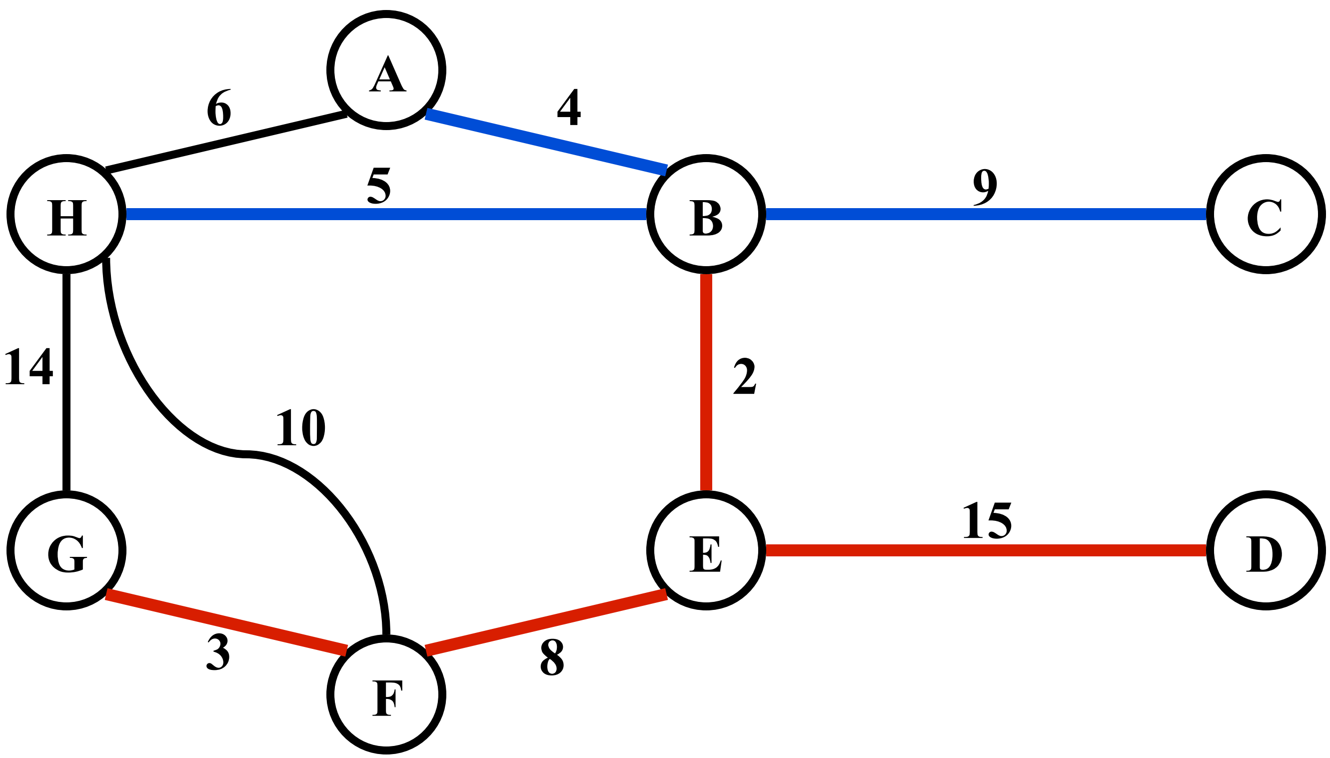
Minimum Spanning Tree

- *Which edges form the minimum spanning tree (MST) of the below graph?*



Minimum Spanning Tree

- Answer:



Minimum Spanning Tree

- MSTs satisfy the *optimal substructure* property. (More on this in Chapter 8.)
  Here: an optimal tree is composed of optimal subtrees
  - Let T be an MST of G with an edge (*u,v*) in the middle
  - Removing (*u,v*) partitions T into two trees T_1 and T_2
  - Claim: T_1 is an MST of $G_1 = (V_1,E_1)$, and T_2 is an MST of $G_2 = (V_2,E_2)$
  - Proof: $w(T) = w(u,v) + w(T_1) + w(T_2)$
    (There can't be a better tree than T_1 or T_2, or T would be suboptimal)

Prim's MST Algorithm

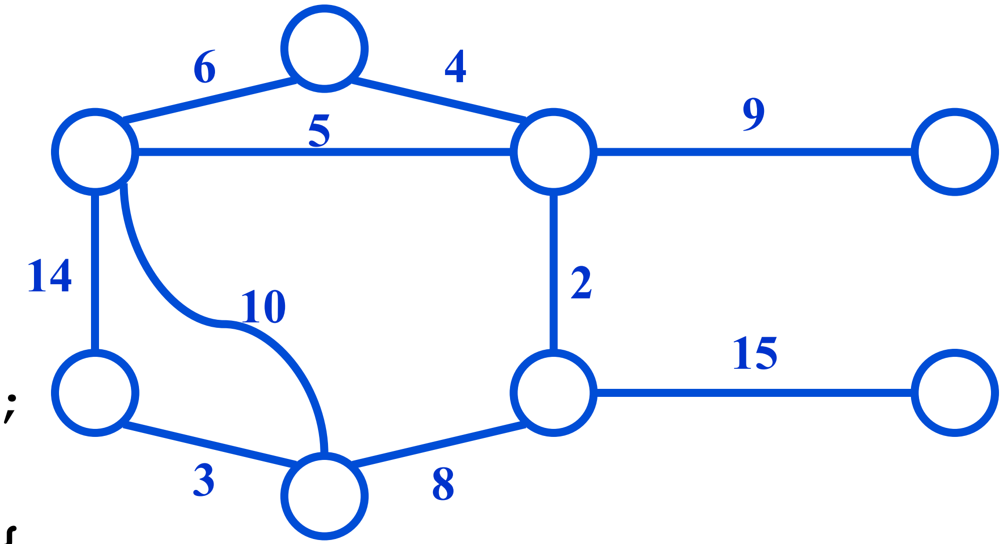
- Greedy strategy:
  - Choose some start vertex as current-tree
  - Greedy rule: Add edge from graph to current-tree that
    - has the lowest weight of edges that…
    - have one vertex in the tree and one not in the tree.
- Thus builds-up one tree by adding a new edge to it
- Can this lead to an infeasible solution?
  (Tell me why not.)
- Is it optimal? (Yes. Need a proof.)

Tracking Edges for Prim's MST

- Candidates edges: edge from a tree-node to a non-tree node
  - Since we'll choose smallest, keep only one candidate edge for each non-tree node
  - But, may need to make sure we always have the smallest edge for each non-tree node
- Fringe-nodes: non-trees nodes adjacent to the tree
- Need data structure to hold fringe-nodes
  - Priority queue, ordered by min-edge weight
  - May need to update priorities!

Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```

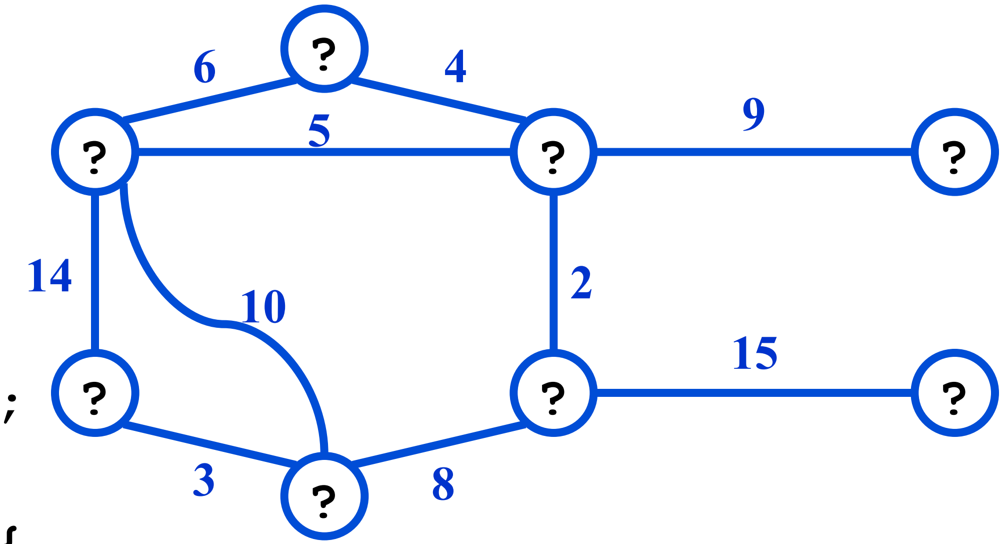


Run on example graph

```
prim(adj,start,parent) { // Textbook's code - compare!
    n = adj.last
    for i = 1 to n
        key[i] = ∞    // key is a local array
    key[start] = 0
    parent[start] = 0
    // the following statement initializes the
    // container h to the values in the array key
    h.init(key,n)
    for i = 1 to n {
        v = h.del()
        ref = adj[v]
        while (ref != null) {
            w = ref.ver
            if (h.isin(w) && ref.weight < h.keyval(w)) {
                parent[w] = v
                h.decrease(w,ref.weight)
            }
            ref = ref.next
        }
    }
}
```

Prim's Algorithm

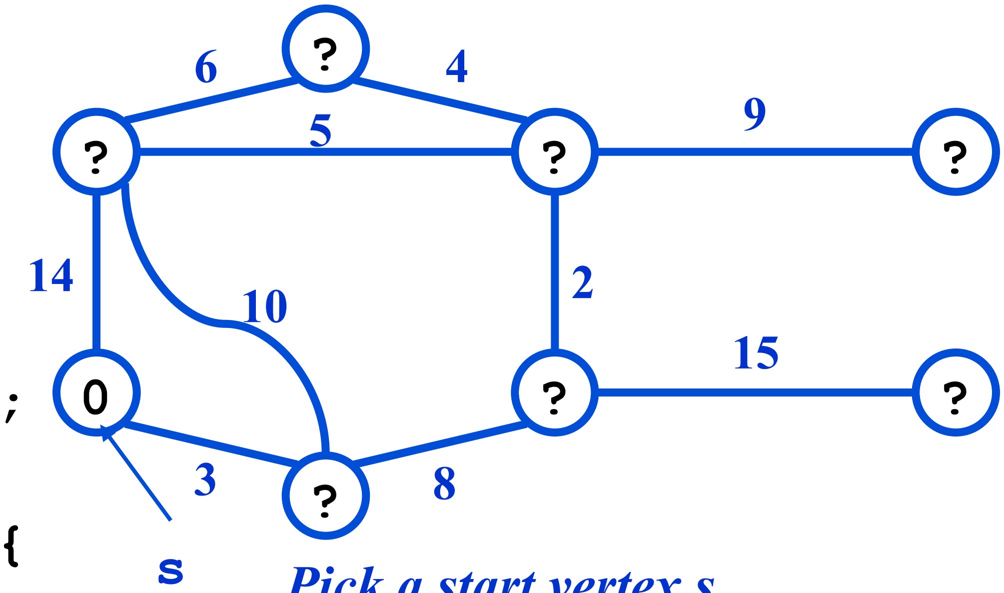
```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



Run on example graph

Prim's Algorithm

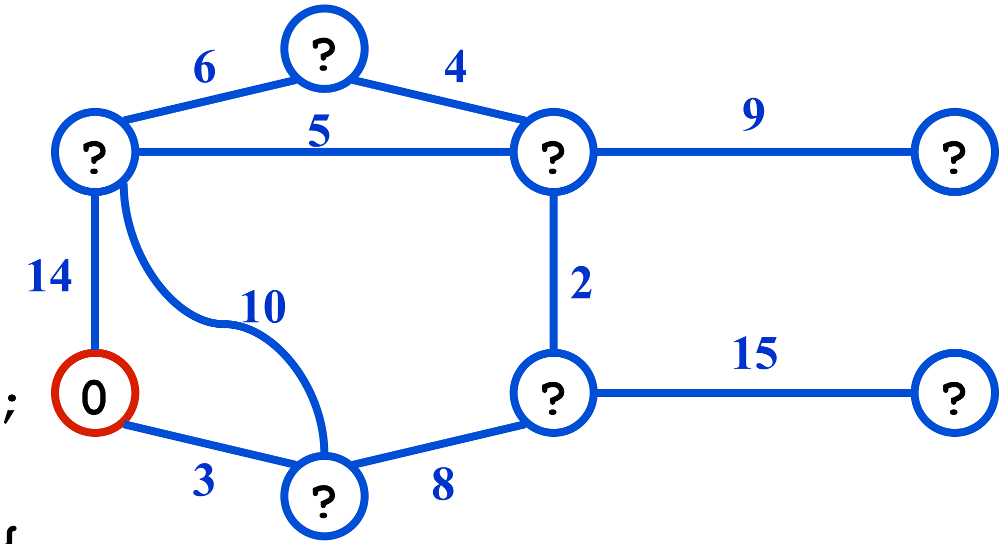
```
MST-Prim(G, wt)
  init PQ to be empty;
  PQ.Insert(s, wt=0);
  parent[s] = NULL;
  while (PQ not empty)
    v = PQ.ExtractMin();
    for each w adj to v
      if (w is unseen) {
         PQ.Insert(w, wt(v,w));
         parent[w] = v;
      }
      else if (w is fringe && wt[v,w] < fringeWt(w)){
         PQ.decreaseKey(w, wt[v,w]);
         parent[w] = v;
      }
```



Pick a start vertex s

Prim's Algorithm

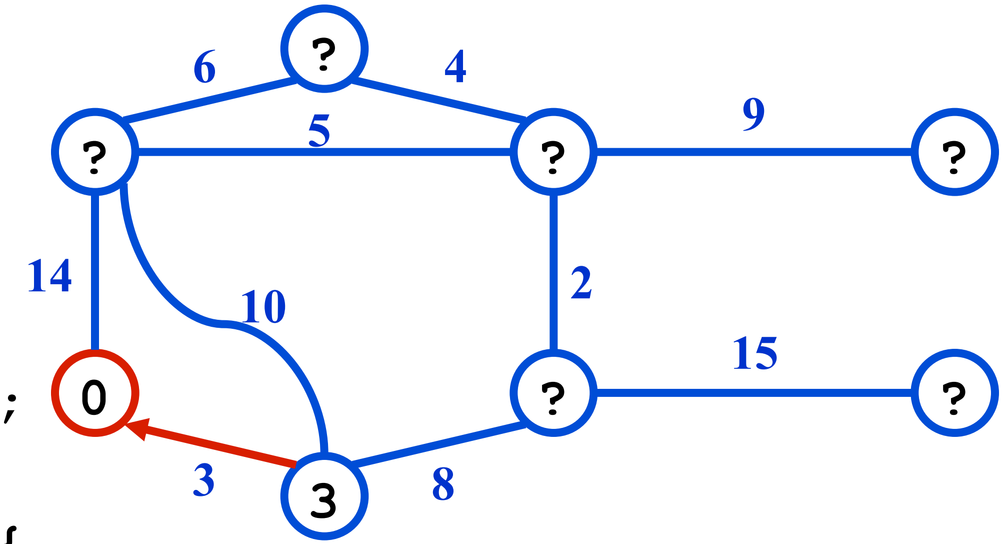
```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



Red vertices have been removed from PQ

Prim's Algorithm

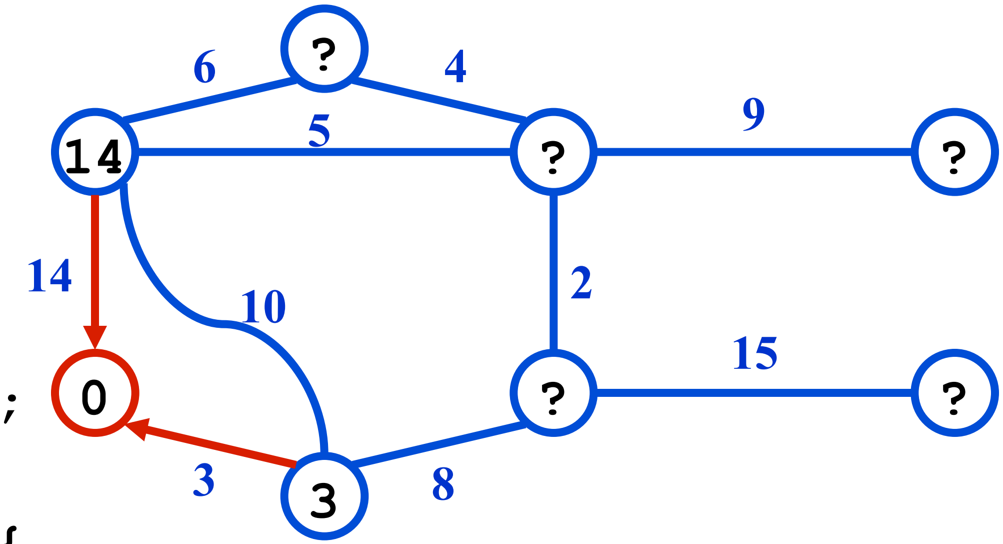
```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



- *Red arrows indicate parent pointers.*
- *Numbers in nodes are fringe weight.*
- *? in node means node is unseen.*

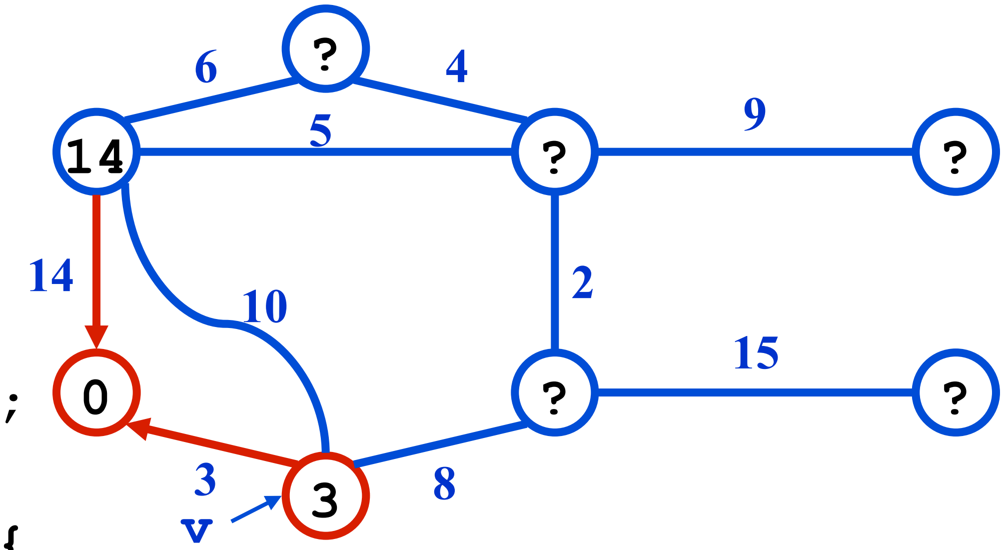
Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



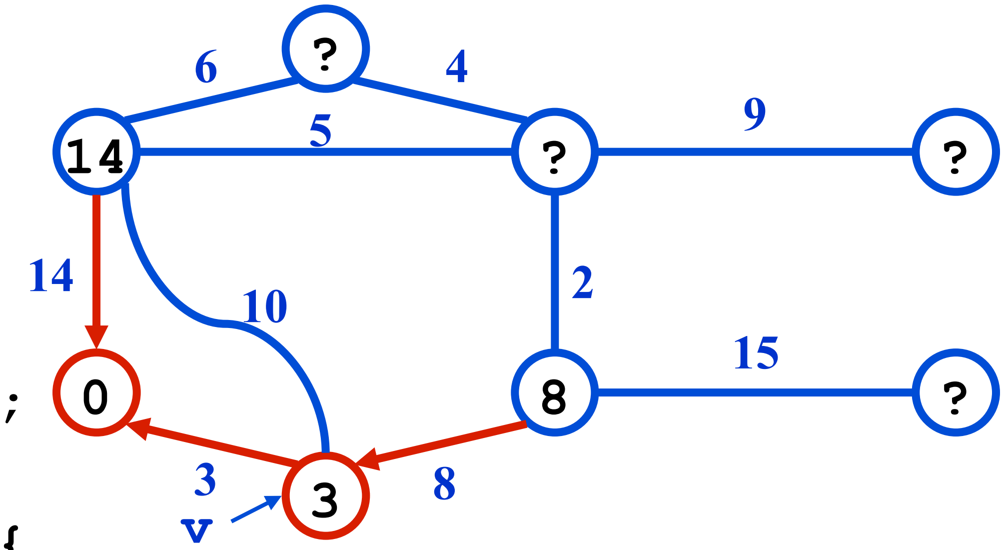
Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



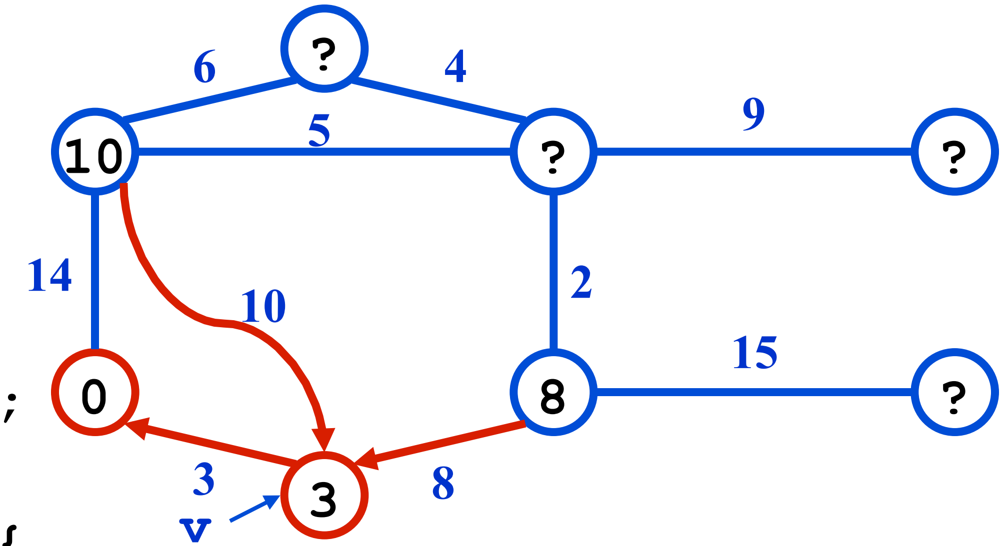
Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



Prim's Algorithm

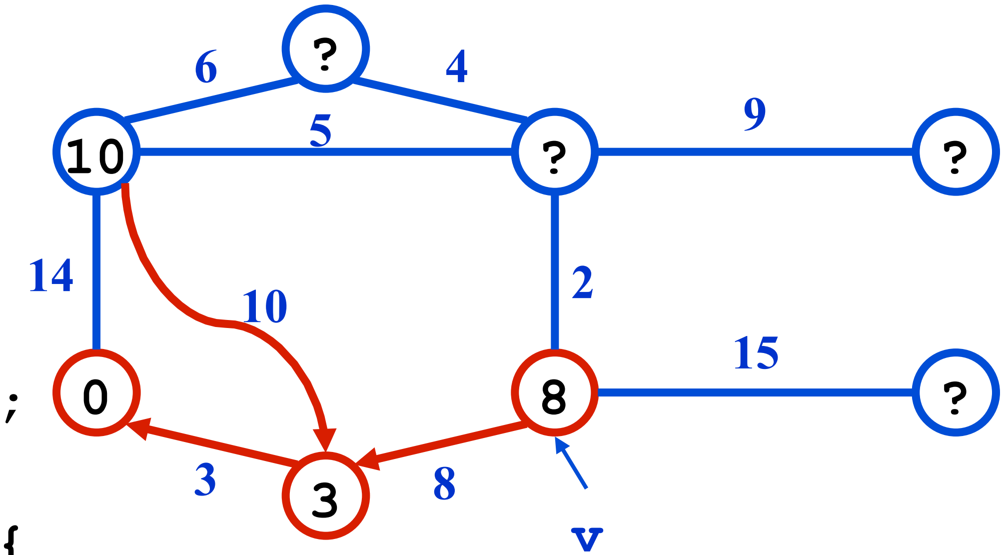
```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



- *Note update of fringe node! FringeWt better, new parent.*

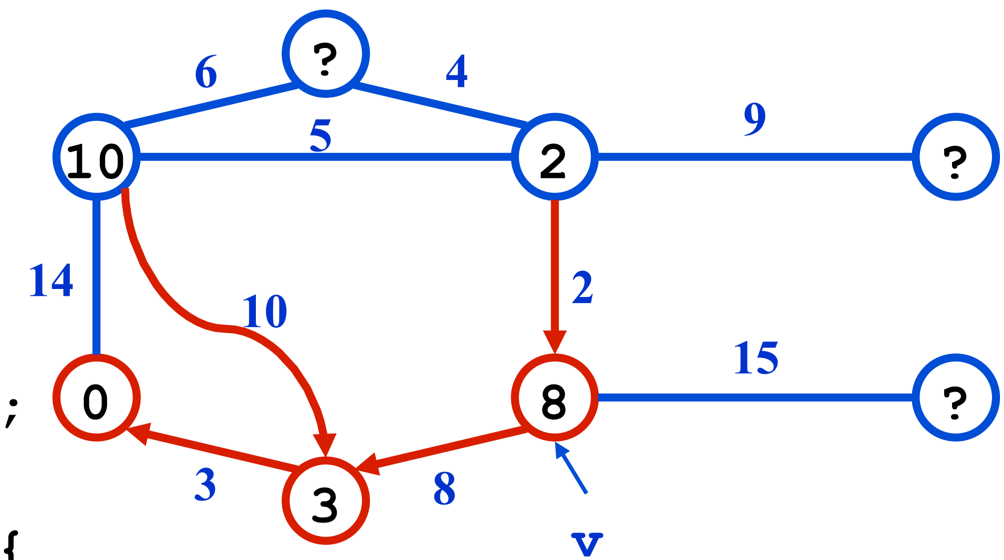
Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



Prim's Algorithm

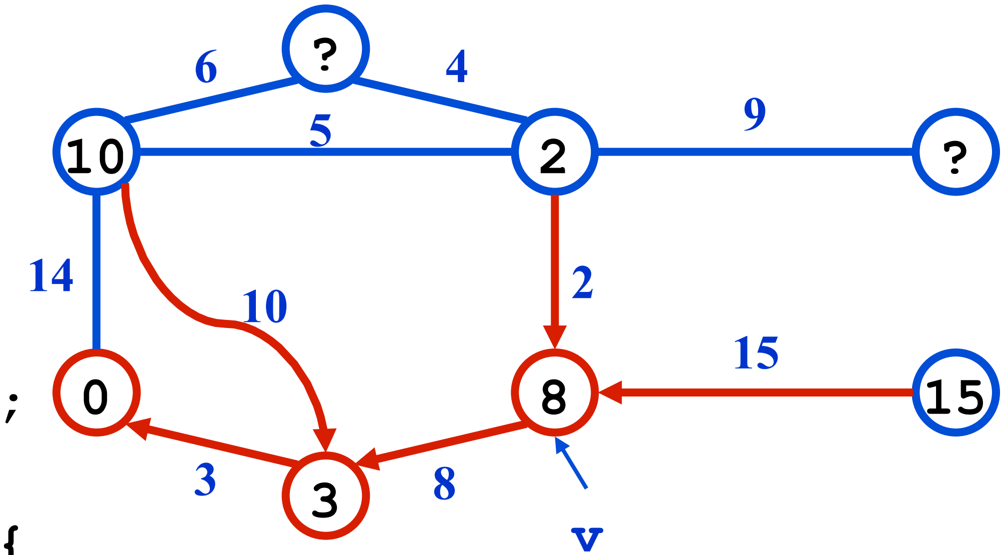
```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



 3/29/10

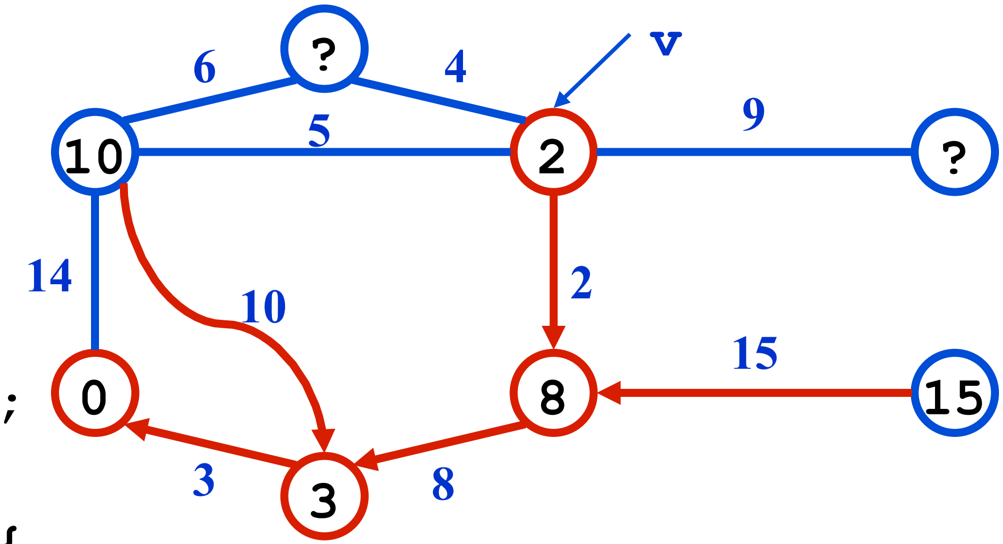
Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



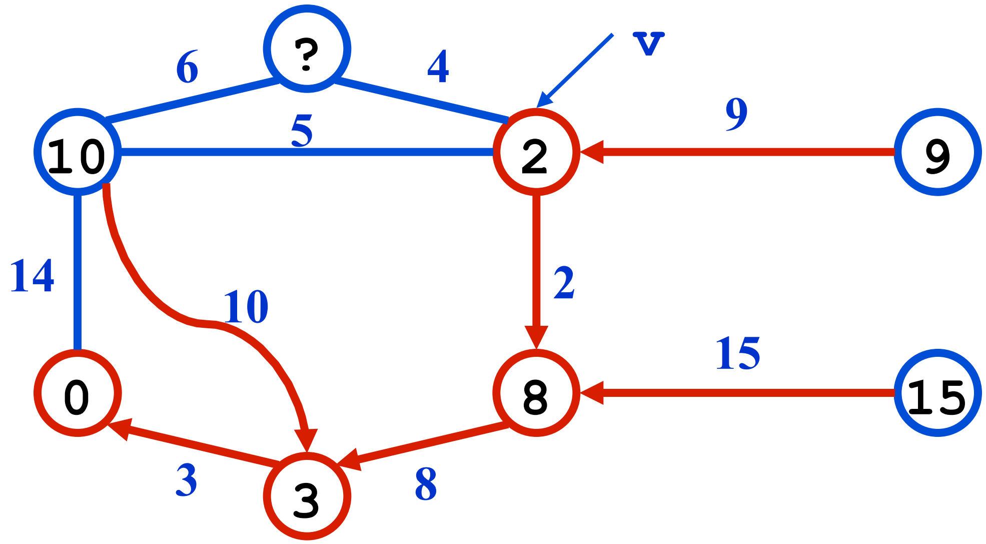
Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



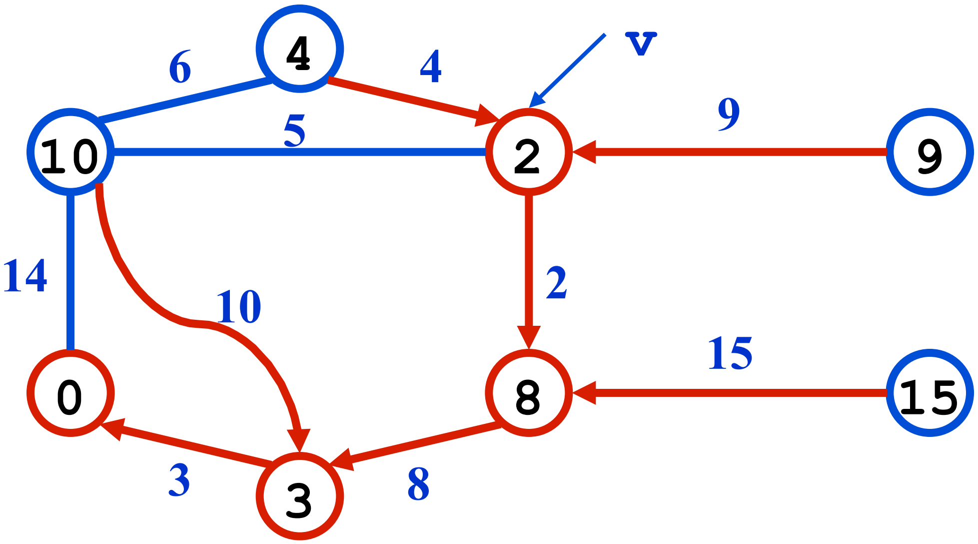
Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



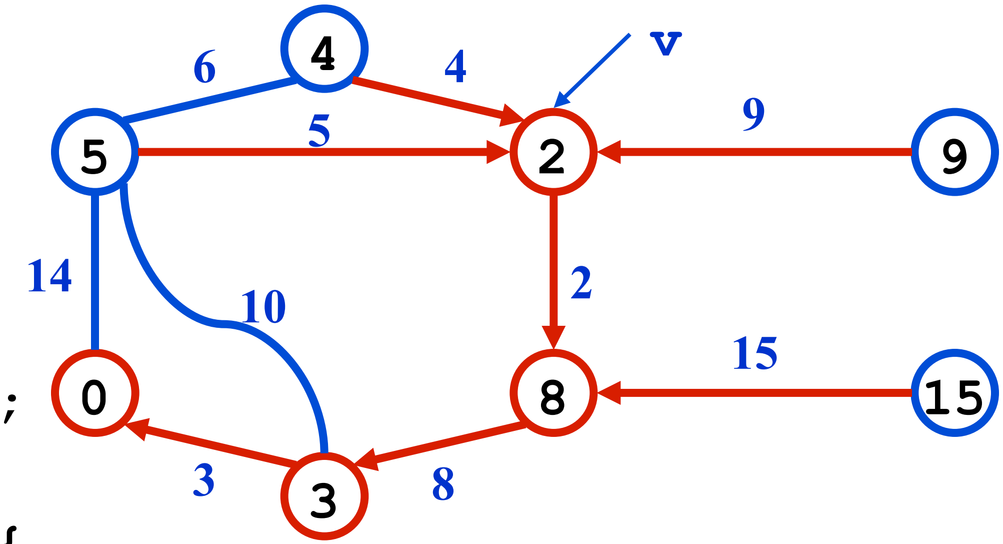
Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



Prim's Algorithm

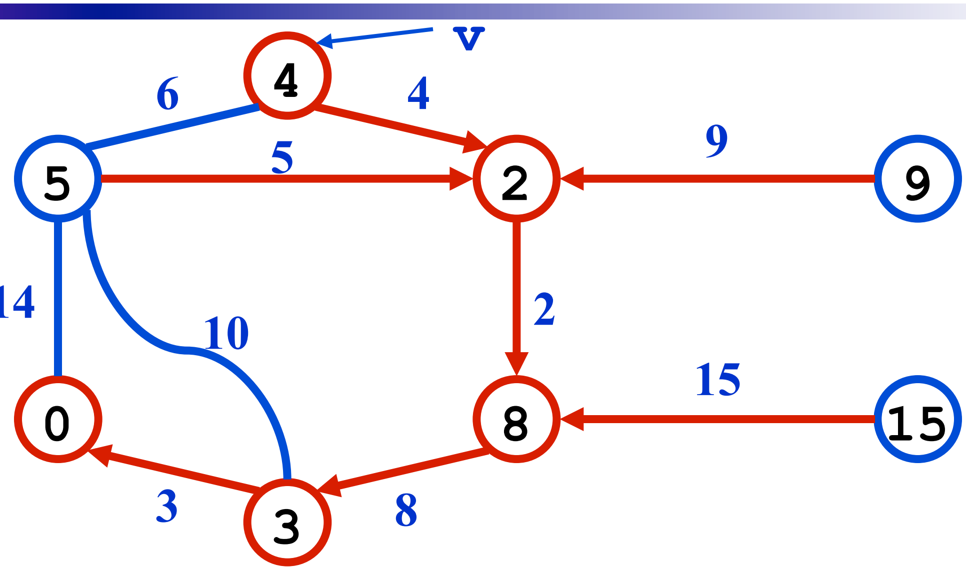
```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



- *Note update of fringe node! FringeWt better, new parent.*

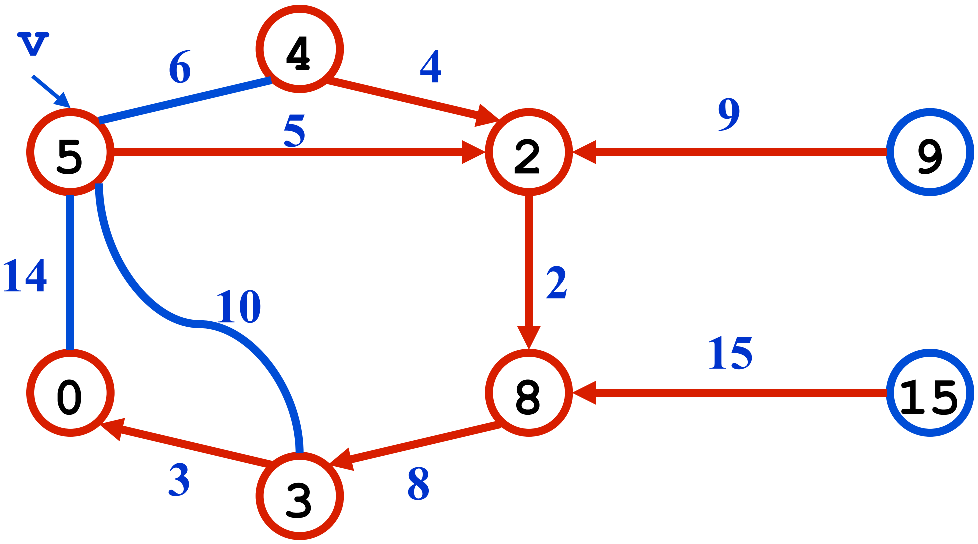
Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



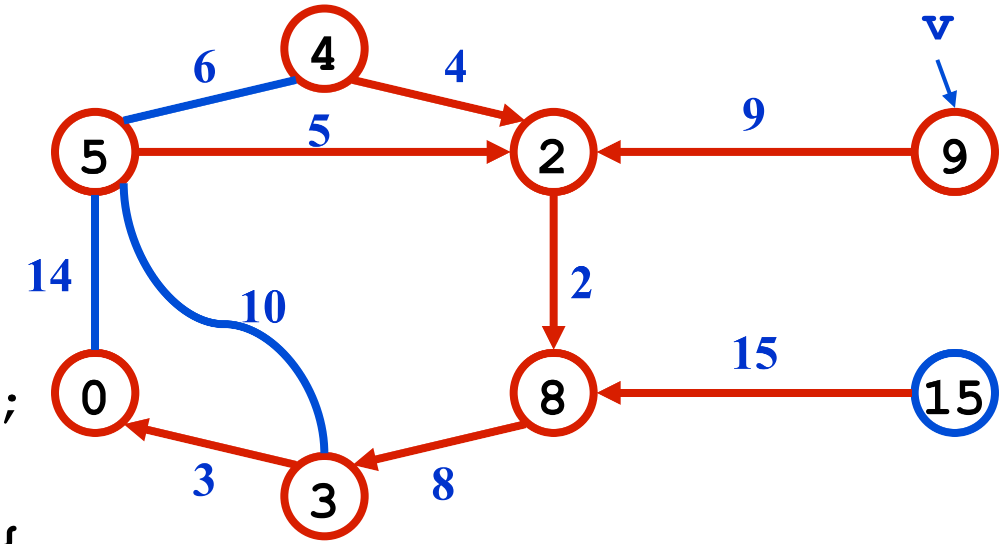
Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



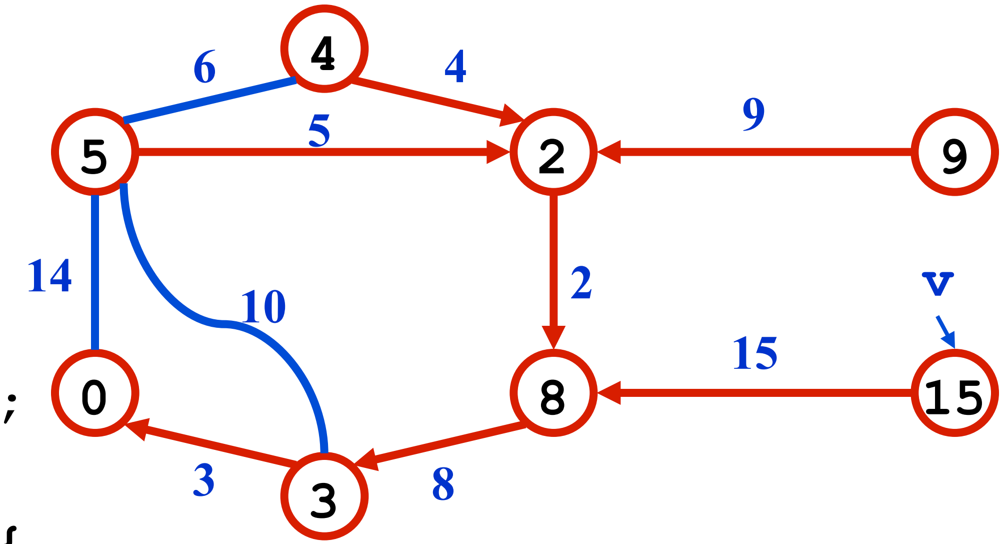
Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```



Cost of Prim's Algorithm

- (Assume connected graph)
- Clearly it looks at every edge, so $\Omega(n+m)$
- Is there more?
  - Yes, priority queue operations
  - ExtractMin called n times
    - How expensive? Depends on the size of the PQ
  - descreaseKey could be called for each edge
    - How expensive is each call?

Worst Case

- If all nodes connected to start, then size of PQ is n-1 right away.
  - Decreases by 1 for each node selected
  - Total cost is O(cost of extractMin for size n-1)
    - Note use of Big-Oh (not Big-Theta)
- Could descreaseKey be called a lot?
  - Yes! Imagine an input that adds all nodes to the PQ at the first step, and then after that calls descreaseKey every possible time. (For you to do.)

Priority Queue Costs and Prim's

- Simplest choice: unordered list
  - PQ.ExtractMin() is just a "findMin"
    - Cost for one call is $\Theta(n)$
    - Total cost for all n calls is $\Theta(n^2)$
  - PQ.decreaseKey() on a node finds it, changes it.
    - Cost for one call is $\Theta(n)$
    - But, if we can index an array by vertex number, the cost would be $\Theta(1)$.
      If so, worst-case total cost is $\Theta(m)$
- Conclusion: Easy to get $\Theta(n^2)$

Better PQ Implementations

- Consider using a min-heap for the Priority Queue
  - PQ.ExtractMin() is O(lg n) each time
    - Called n times, so like Heap's Construct: efficient!
  - What about PQ.decreaseKey() ?
- Our need: given a vertex-ID, change the value stored
  - But our basic heap implementation does not allow look-ups based on vertex-ID!
- Solution: Indirect heaps (see pages 142-145)
  - Heap structure stores indices to data in an array that doesn't change
  - Can increase or decrease key in O(lg n) after O(1) lookup

Better PQ Implementations (2)

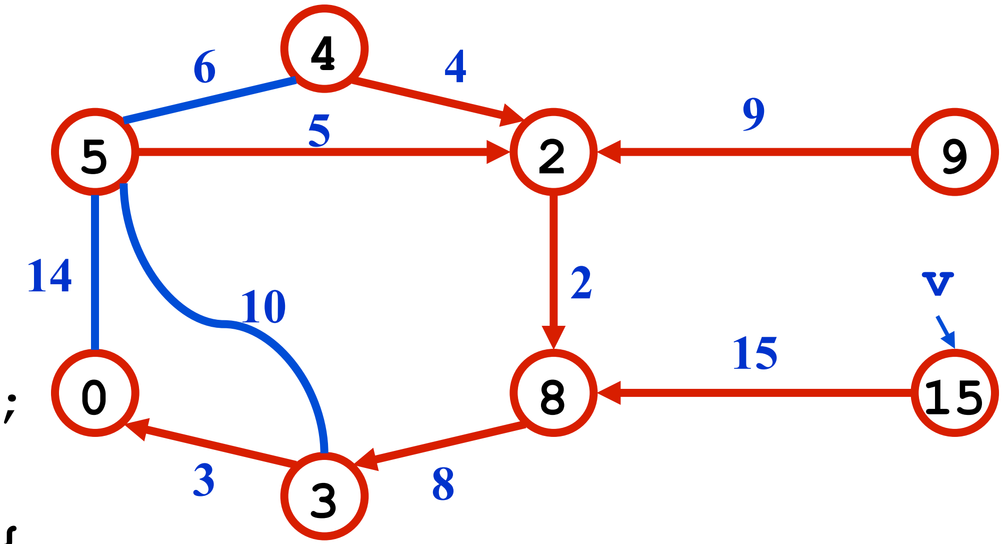
- Use Indirect Heaps for the PQ
  - PQ.decreaseKey() is O(lg n) also
    - Called for each edge encountered in MST algorithm
    - So O(m x lg n)
    - Overall: Might be better $\Theta(n^2)$ than if $m << n^2$
- Fibonacci heaps: an even more efficient PQ implementation. We won't cover these.
  - $\Theta(m + n \lg n)$

Kruskal's MST Algorithm

- Prim's approach:
  - Build one tree. Make the one tree bigger and as good as it can be.
- Kruskal's approach
  - Choose the best edge possible: smallest weight
  - Not one tree – maintain a forest!
  - Each edge added will connect two trees.
    Can't form a cycle in a tree!
  - After adding n-1 edges, you have one tree, the MST

Prim's Algorithm

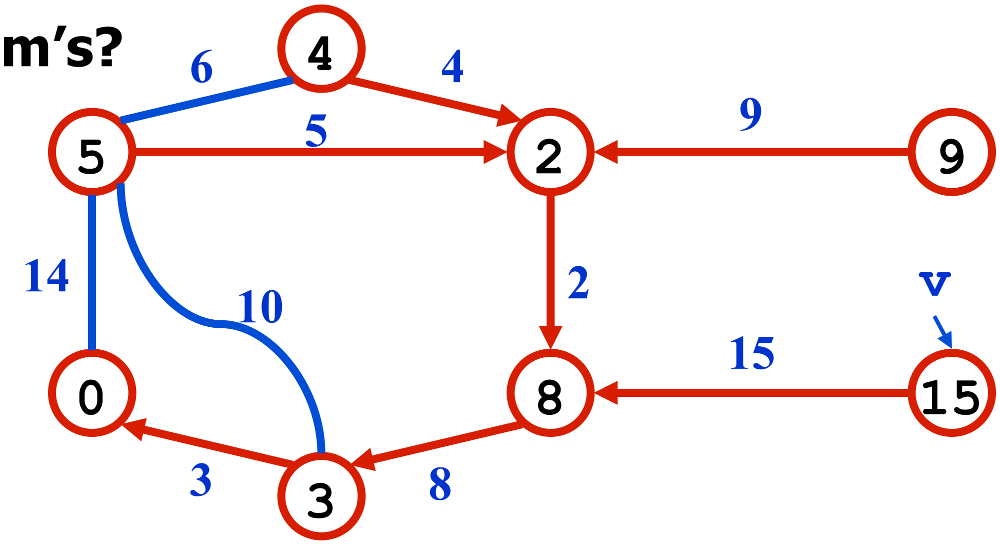
```
MST-Prim(G, wt)
  init PQ to be empty;
  PQ.Insert(s, wt=0);
  parent[s] = NULL;
  while (PQ not empty)
    v = PQ.ExtractMin();
    for each w adj to v
      if (w is unseen) {
         PQ.Insert(w, wt(v,w));
         parent[w] = v;
      }
      else if (w is fringe && wt[v,w] < fringeWt(w)){
         PQ.decreaseKey(w, wt[v,w]);
         parent[w] = v;
      }
```



3/29/10

Kruskal's Algorithm

Builds the same tree as Prim's?



Sorted list of edges:

- **DC 2**
- **AB 3**
- **DF 4**
- **DE 5**
- **EF 6**
- **CB 8**
- **DG 9**
- **BE 10**
- **AE 14**
- **CH 15**

Strategy for Kruskal's

- EL = sorted set of edges ascending by weight
- Foreach edge e in EL
  - T1 = tree for head(e)
  - T2 = tree for tail(e)
  - If (T1 != T2)
    - add e to the output (the MST)
    - Combine trees T1 and T2
- Seems simple, no?
  - But, how do you keep track of what trees a node is in?
  - Trees are sets. Need to findset(v) and "union" two sets

```
kruskal(edgelist,n) {
   sort(edgelist)
   for i = 1 to n
      makeset(i)
   count = 0
   i = 1
   while (count < n - 1) {
      if (findset(edgelist[i].v) !=
                 findset(edgelist[i].w)) {
         println(edgelist[i].v + " "
                  + edgelist[i].w)
            count = count + 1
            union(edgelist[i].v,edgelist[i].w)
      }
      i = i + 1
   }
}
```

Union/Find and Disjoint Sets

- See Section 3.6, page 150-161
- Sets stored as a parent array (see bottom of p. 151)
  - findset(v): trace upward in parent array
  - union(i,j): make one tree a child of a node it the other
- Improvements! E.g. path compression
  - O(lg m)

Complexity for Kruskal's

- Overall: $\Theta(m \lg m)$

Single Source Shortest Path

- Problem: Given a node v, find the minimum distance from v to *either* another node w *or* to all other nodes,
  where distance is the sum of the edge-weights on the path
- A solution: Dijkstra's algorithm
  - Who's Dijkstra? See class wall-of-fame!

Dijkstra's Shortest Path Algorithm

- Identical *in structure* to Prim's MST algorithm
  - Of course it solves a different problem!
  - Same time complexity
- Additional input parameter(s)
  - Start node v
  - Destination node w (if needed)
- Different output: a path from v to w and a cost (or sets of paths and costs)
  - The tree is the sets of shortest paths to nodes
- Different greedy strategy:
  - Store shortest paths to fringe-nodes in priority queue
  - Store path-distance to node, not just the one edge-weight

Reminder: Prim's Algorithm

```
MST-Prim(G, wt)
 init PQ to be empty;
 PQ.Insert(s, wt=0);
 parent[s] = NULL;
 while (PQ not empty){
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        PQ.Insert(w, wt(v,w));
        parent[w] = v;
     }
     else if (w is fringe && wt[v,w] < fringeWt(w)){
        PQ.decreaseKey(w, wt[v,w]);
        parent[w] = v;
     }
```

Dijkstra' Algorithm

```
dijkstra(G, wt, s)
 init PQ to be empty;
 PQ.Insert(s, dist=0);
 parent[s] = NULL; dist[s] = 0;
 while (PQ not empty)
   v = PQ.ExtractMin();
   for each w adj to v
     if (w is unseen) {
        dist[w] = dist[v] + wt(v,w)
        PQ.Insert(w, dist[w] );
        parent[w] = v;
      }
     else if (w is fringe && dist[v] + wt(v,w) < dist[w] )
   {
        dist[w] = dist[v] + wt(v,w)
        PQ.decreaseKey(w, dist[w]);
        parent[w] = v;
     }
```

Notes on Dijkstra's Algorithm

- Use dist[] to store distances from start to any fringe or tree node
- Store and calculate using distances instead of edge-weights (like in Kruskal's MST)
- What's the output?
  - Tree captured in the parent[] array
  - Shortest distance to each node in dist[] array
  - Trace shortest path in reverse by using parent[] to move from target back to start node, s

```
dijkstra(adj, start, parent) {
    n = adj.last
    for i = 1 to n { key[i] = ∞}  // key is a local array
    key[start] = 0;   predecessor[start] = 0
    // the following statement initializes the
    // container h to the values in the array key
    h.init(key,n)
    for i = 1 to n {
            v = h.min_weight_index()
            min_cost = h.keyval(v)
            v = h.del()
            ref = adj[v]
            while (ref != null) {
                        w = ref.ver
                        if (h.isin(w) && min_cost + ref.weight < h.keyval(w)) {
                            predecessor[w] = v
                            h.decrease(w, min_cost+ref.weight)
                        } // end if
                        ref = ref.next
            } // end while
    } // end for
}
```

Correctness of These Greedy Algorithms

- Recall that the greedy approach may or may not guarantee an optimal result
- Do these produce optimal solutions?
  - The min weight spanning tree? Kruskal's, Prim's
  - The shortest path from s? Dijkstra's
- Answer: Yes, they do.
  - Proofs in the text
  - Proofs by induction, also using proof by contradiction