# CS583 Lecture 09

Jana Kosecka

Graph Algorithms
Topological Sort
Strongly Connected Component
Minimum Spanning Tree

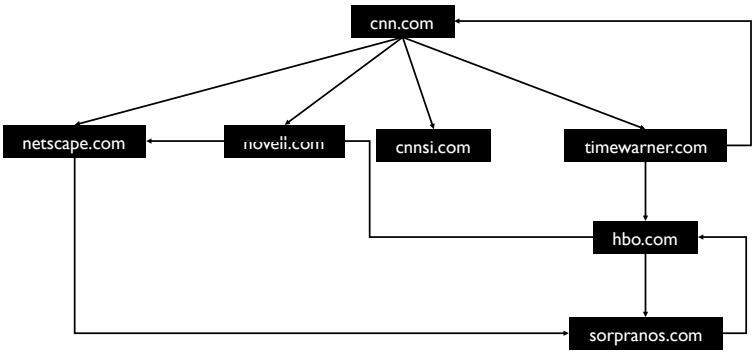
Many slides here are based on E. Demaine , D. Luebke, Kleinberg-Tardos slides

## Graph Algs. Continued

- Review BFS
- Application of BSF – check bipartiteness
- Review DFS
- Check for cycles

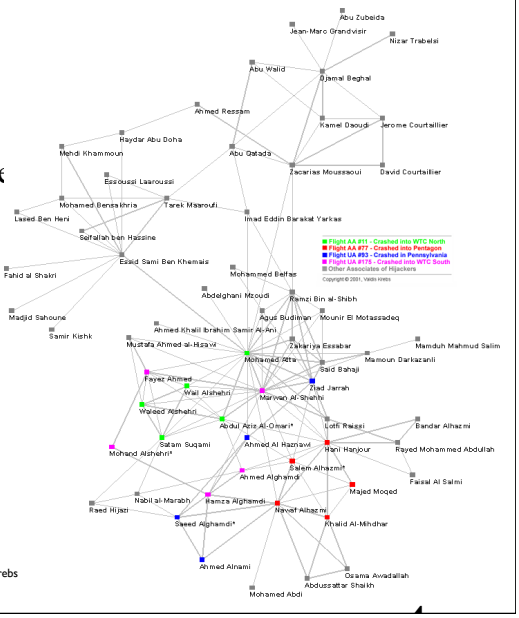
# World Wide Web

•Web graph.
  Node: web page.
  Edge: hyperlink from one page to another.




# 9-11 Terrorist Network

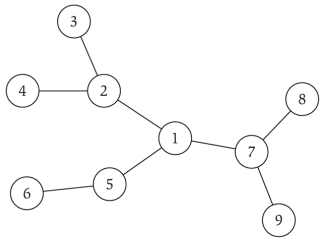
•Social network graph.
  Node: people.
  Edge: relationship betwe



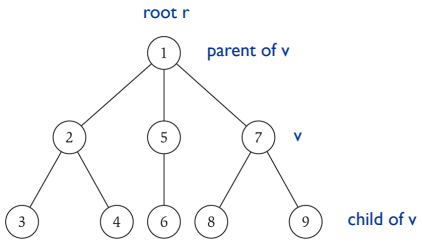

Reference: Valdis Krebs, http://www.firstmonday.org/issues/issue7_4/krebs

# Rooted Trees

•Rooted tree. Given a tree T, choose a root node r and orient each edge away from r.
•Importance. Models hierarchical structure.



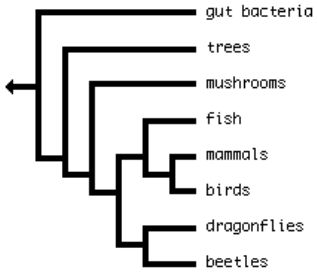

a tree




the same tree, rooted at 1

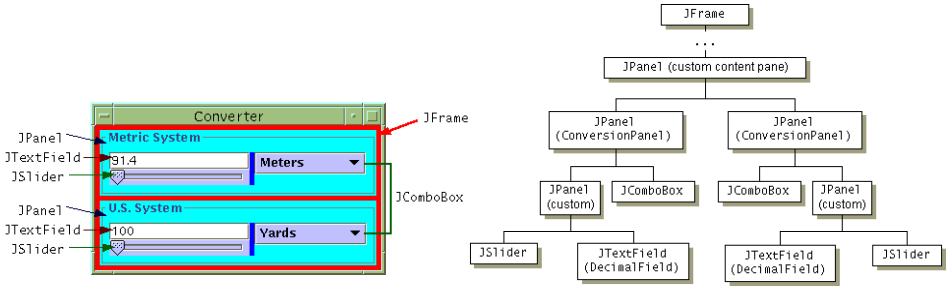
# Phylogeny Trees

•Phylogeny trees. Describe evolutionary history of species.



# GUI Containment Hierarchy

•GUI containment hierarchy. Describe organization of GUI widgets.




Reference: http://java.sun.com/docs/books/tutorial/uiswing/overview/anatomy.html

# Breadth-First Search

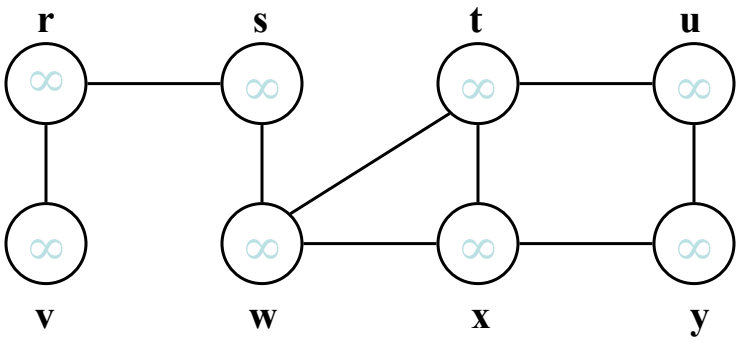
- Again will associate vertex "colors" to guide the algorithm
  - White vertices have not been discovered
  - All vertices start out white
  - Grey vertices are discovered but not fully explored
  - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
  - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

# Breadth-First Search

```
BFS(G, s) {
    initialize vertices;
    Q = {s};        // Q is a queue (duh); initialize
  to s
    while (Q not empty) {
        u = RemoveTop(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

What does `v->d` represent?
What does `v->p` represent?

# Breadth-First Search: Example



## BSF

- Check for bi-partite graphs
- Graphs representing relationships
- All nodes can belong to two subsets
- There are no edges between subsets

## Depth-First Search

- *Depth-first search* is another strategy for exploring a graph
- Explore "deeper" in the graph whenever possible
- Edges are explored out of the most recently discovered vertex *v* that still has unexplored edges
- When all of *v*'s edges have been explored, backtrack to the vertex from which *v* was discovered

# Depth-First Search

- Vertices initially colored white
- Then colored gray when discovered
- Then black when finished

# Depth-First Search: The Code

```
DFS(G)
{
   for each vertex u ∈ G->V
   {
      u->color = WHITE;
   }
   time = 0;
   for each vertex u ∈ G->V
   {
      if (u->color ==
  WHITE)
         DFS_Visit(u);
   }
}
```

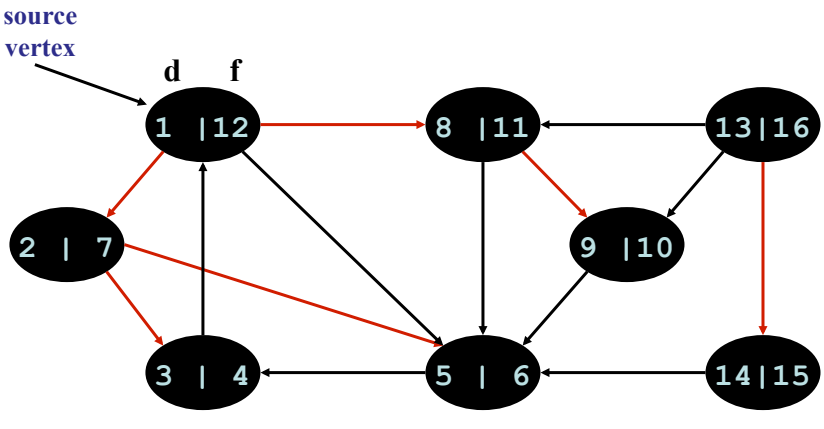
```
DFS_Visit(u)
{
   u->color = GREY;
   time = time+1;
   u->d = time;
   for each v ∈ u->Adj[]
   {
      if (v->color ==
  WHITE)
         DFS_Visit(v);
   }
   u->color = BLACK;
   time = time+1;
   u->f = time;
}
```

# Depth-First Sort Analysis

- This running time argument is an informal example of *amortized analysis*

- "Charge" the exploration of edge to the edge:
- Each loop in DFS_Visit can be attributed to an edge in the graph
- Runs once/edge if directed graph, twice if undirected
- Thus loop will run in O(E) time, algorithm O(V+E)
- Considered linear for graph, b/c adj list requires O(V+E) storage
- Important to be comfortable with this kind of reasoning and analysis

# DFS Example

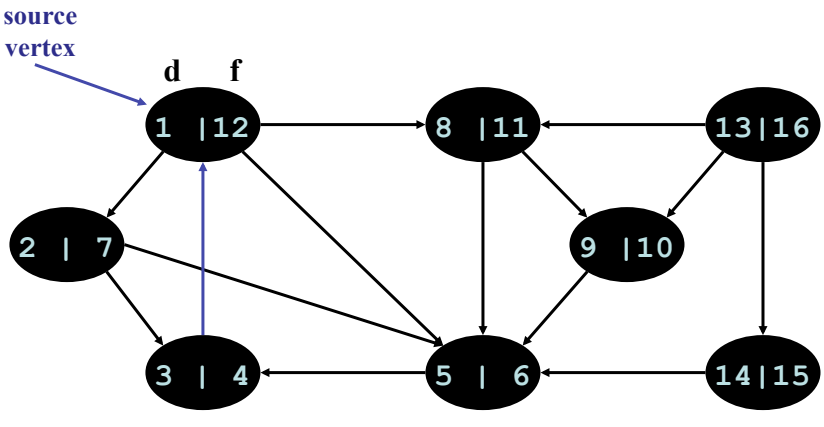


# DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:

- *Tree edge*: encounter new (white) vertex
- *Back edge*: from descendent to ancestor
  Encounter a grey vertex (grey to grey)

# DFS Example



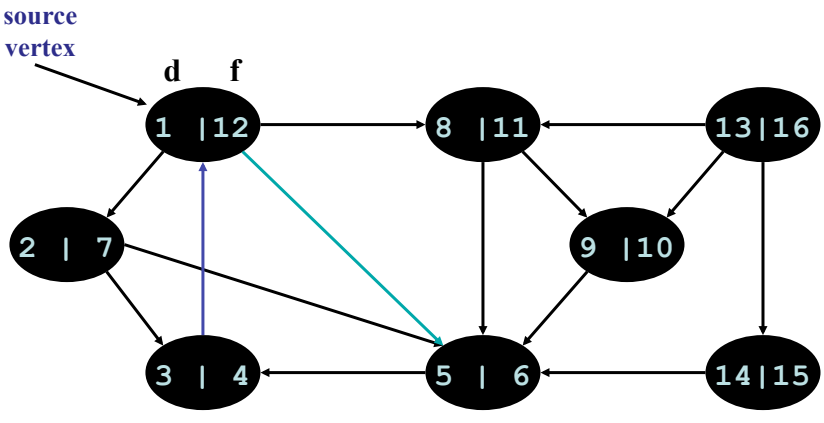

**Tree edges** **Back edges**

# DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  *Tree edge*: encounter new (white) vertex
  *Back edge*: from descendent to ancestor
  *Forward edge*: from ancestor to descendent

  Not a tree edge, though
  From grey node to black node

# DFS Example



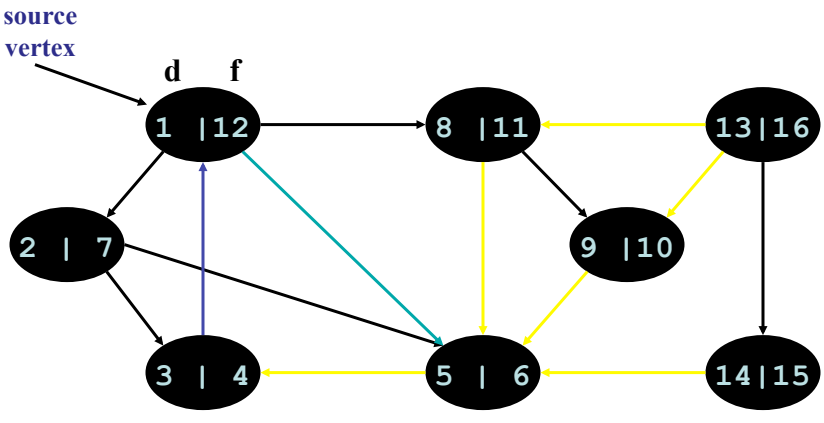

**Tree edges** **Back edges** **Forward edges**

# DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  *Tree edge*: encounter new (white) vertex
  *Back edge*: from descendent to ancestor
  *Forward edge*: from ancestor to descendent
  *Cross edge*: between a tree or subtrees

  From a grey node to a black node

# DFS Example




**Tree edges** **Back edges** **Forward edges** **Cross edges**

## DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  *Tree edge*: encounter new (white) vertex
  *Back edge*: from descendent to ancestor
  *Forward edge*: from ancestor to descendent
  *Cross edge*: between a tree or subtrees
- Note: tree & back edges are important; most algorithms don't distinguish forward & cross

## DFS: Kinds Of Edges

- Thm 23.9 (22.10 – in 3rd edition): If G is undirected, a DFS produces only tree and back edges
- Suppose you have $u.d < v.d$
- Then search discovered u before v, so first time v is discovered it is white – hence the edge *(u,v)* is a tree edge
- Otherwise the search already explored this edge in direction from *v to u*
- edge must actually be a back edge since
- *u* is still gray

# DFS And Graph Cycles

- Thm: An undirected graph is *acyclic* iff a DFS yields no back edges
- If acyclic, no back edges (because a back edge implies a cycle
- If no back edges, acyclic
  - No back edges implies only tree edges (*Why?*)
  - Only tree edges implies we have a tree or a forest
  - Which by definition is acyclic
- Thus, can run DFS to find whether a graph has a cycle

# DFS And Cycles

- *How would you modify the code to detect cycles?*

```
DFS(G)
{
   for each vertex u ∈ G->V
   {
      u->color = WHITE;
   }
   time = 0;
   for each vertex u ∈ G->V
   {
      if (u->color == WHITE)
         DFS_Visit(u);
   }
}
```

```
DFS_Visit(u)
{
   u->color = GREY;
   time = time+1;
   u->d = time;
   for each v ∈ u->Adj[]
   {
      if (v->color == WHITE)
         DFS_Visit(v);
   }
   u->color = BLACK;
   time = time+1;
   u->f = time;
}
```

# DFS And Cycles

- *What will be the running time ?*

```
DFS(G)
{
   for each vertex u ∈ G->V
   {
      u->color = WHITE;
   }
   time = 0;
   for each vertex u ∈ G->V
   {
      if (u->color == WHITE)
         DFS_Visit(u);
   }
}
```

```
DFS_Visit(u)
{
   u->color = GREY;
   time = time+1;
   u->d = time;
   for each v ∈ u->Adj[]
   {
      if (v->color == WHITE)
         DFS_Visit(v);
   }
   u->color = BLACK;
   time = time+1;
   u->f = time;
}
```

# DFS And Cycles

- *What will be the running time?*
- A: O(V+E)
- We can actually determine if cycles exist in O(V) time:

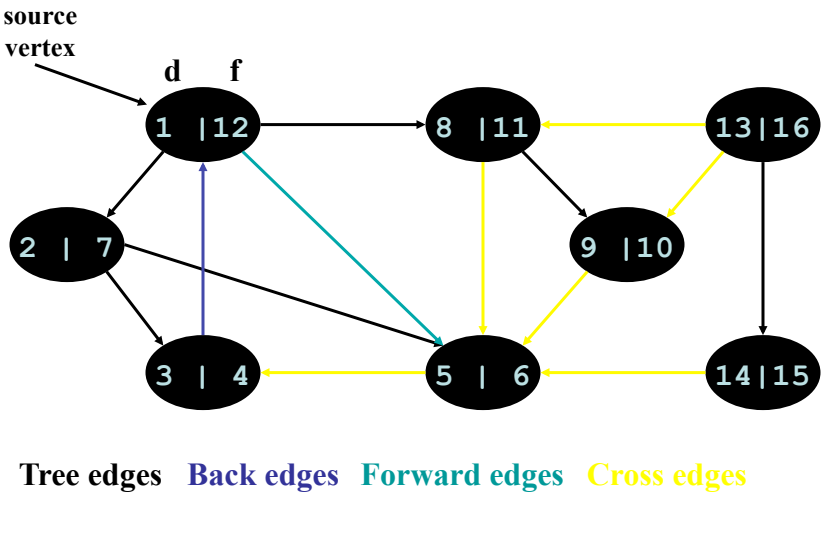
  In an undirected acyclic forest, $|E| \leq |V| - 1$

  So count the edges: if ever see |V| distinct edges, must have seen a back edge along the way

# Review: Kinds Of Edges

- Thm: If G is undirected, a DFS produces only tree and back edges
- Thm: An undirected graph is *acyclic* iff a DFS yields no back edges
- Thm, can run DFS to find cycles

# Review: Kinds of Edges



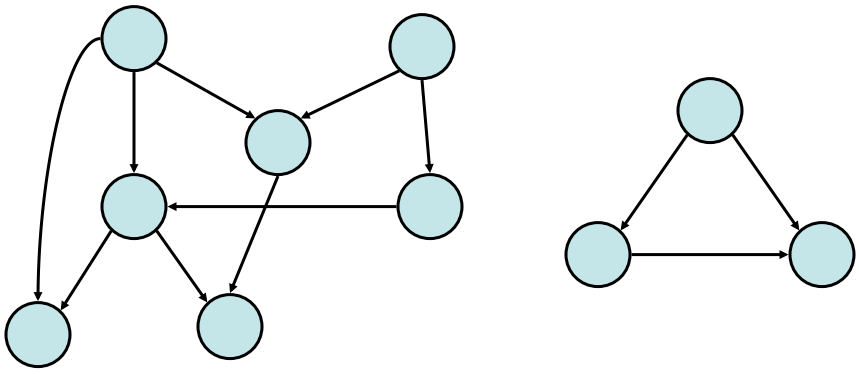
# DFS And Cycles

- Running time: O(V+E)
- We can actually determine if cycles exist in O(V) time:
  In an undirected acyclic forest, $|E| \leq |V| - 1$
  So count the edges: if ever see |V| distinct edges, must have seen a back edge along the way
  *Why not just test if |E| <|V| and answer the question in constant time?*

  We can have some isolated component nodes not connected by any edges to the rest of the graph

# Directed Acyclic Graphs

- A *directed acyclic graph* or *DAG* is a directed graph with no directed cycles:



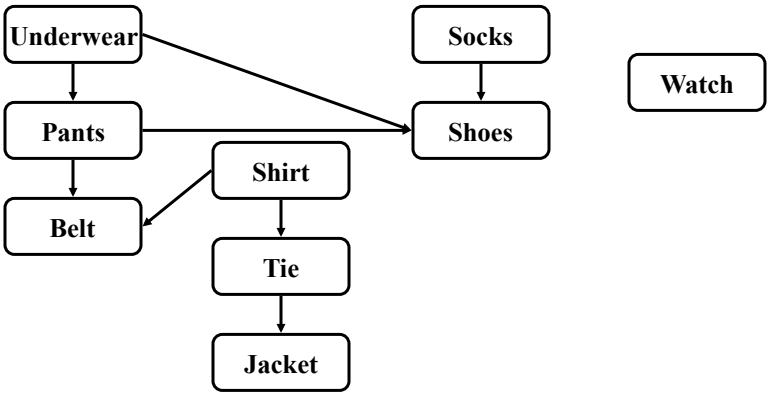
# DFS and DAGs

- Argue that a directed graph G is acyclic iff a DFS of G yields no back edges:
- Forward: if G is acyclic, will be no back edges
  Trivial: a back edge implies a cycle
- Backward: if no back edges, G is acyclic
- Argue contrapositive: Suppose G has a cycle ⇒ we will show that DFS will produce a back edge
- Let $v$ be the vertex on the cycle first discovered, and $u$ be the predecessor of $v$ on the cycle
- When $v$ discovered, whole cycle is white
- Must visit everything reachable from $v$ before returning from DFS-Visit()
- So path from u→v is descendant of v hence (gray→gray), thus $(u, v)$ is a back edge

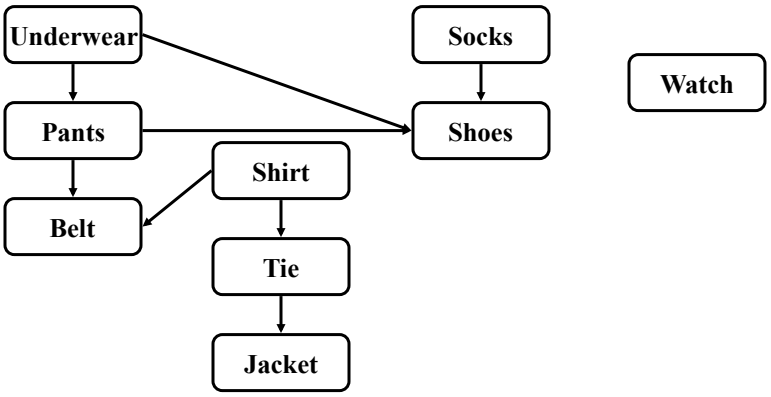
# Topological Sort

- *Topological sort* of a DAG:
  Linear ordering of all vertices in graph G such that
  vertex $u$ comes before vertex $v$ if edge $(u, v) \in$ G
- Real-world example: getting dressed

## Getting Dressed




## Getting Dressed




# Topological Sort Algorithm

```
Topological-Sort()
{
  Run DFS
  When a vertex is finished, output it
  On the front of linked list
  Vertices are output in reverse
   topological order
}
```

- Time: O(V+E)
- Correctness: Want to prove that

  $(u,v) \in G \Rightarrow u{\rightarrow}f > v{\rightarrow}f$

# Topological Sort

- Ordering of activities in the presence of constraints
- Process scheduling

## Correctness of Topological Sort

- Claim: $(u,v) \in G \Rightarrow u\rightarrow f > v\rightarrow f$
- Show that if there is an edge from u to v, finishing time of u is greater then v

- When $(u,v)$ is explored, $u$ is gray
  $v$ = gray $\Rightarrow$ $(u,v)$ is back edge. Contradiction (*Why?*)
  hence v cannot be gray – since there are no cycles
  $v$ = white $\Rightarrow$ $v$ becomes descendent of $u$ $\Rightarrow$ $v\rightarrow f < u\rightarrow f$
  (since must finish $v$ before backtracking and finishing $u$)
  $v$ = black $\Rightarrow$ $v$ already finished $\Rightarrow$ $v\rightarrow f < u\rightarrow f$

## Connected components

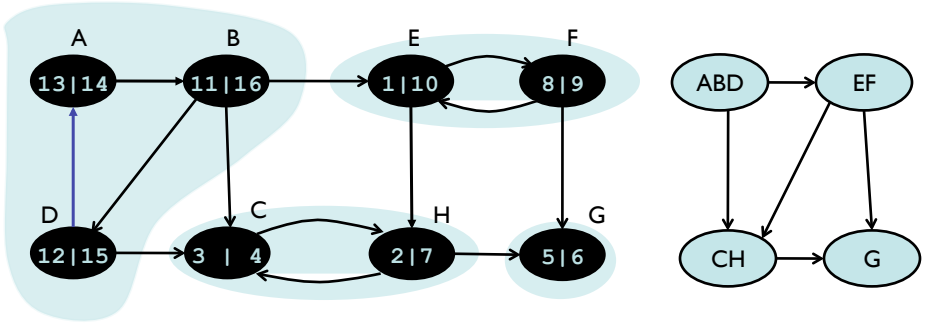
- Undirected graph is connected: there is a path from u -> v For all u and v

- Directed graph is strongly connected if there is a path from u->v for all u and v

- How can we find strongly connected components of a directed graph

# Strongly Connected components

- SSC algorithm will induce component graph which is a DAG

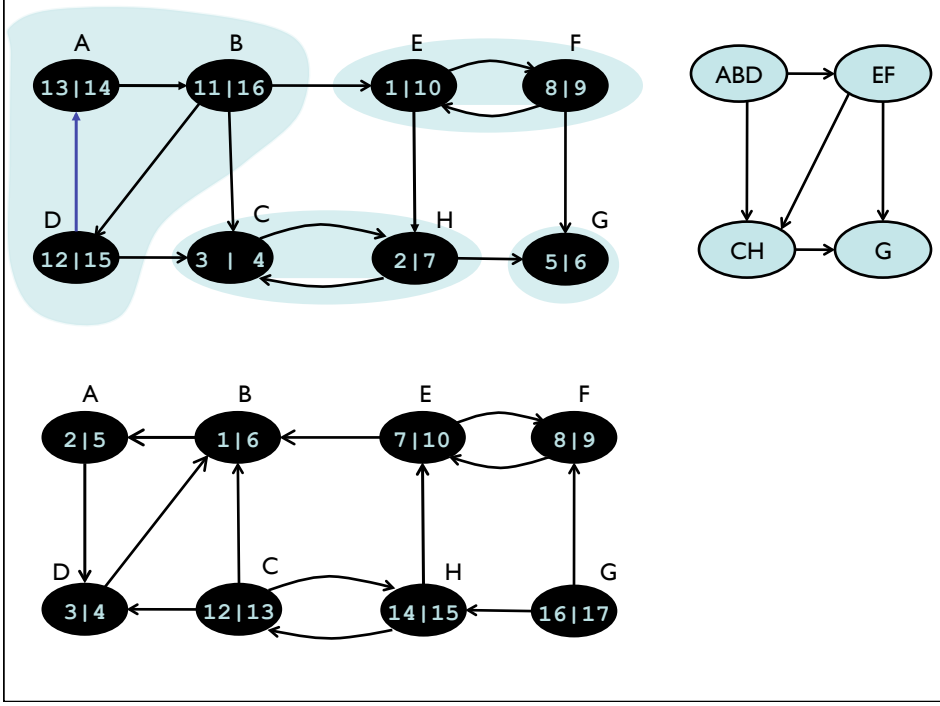



# Strongly Connected component

- How to use DFS to find strongly connected component
- When running DFS_visit recursively it will stop once all nodes reachable from start are visited – as a result you will have one DFS tree.
- Observation DFS tree will contain one of more strongly connected components
- How to run DFS so it will break (finish DFS_visit) just after finishing only one SCC

# Strongly Connected Components

- Call DFS to compute finishing times *f[u]* of each vertex
- Create transpose graph (directions of edges reversed)
- Call DFS on the transpose, but in the main loop of DFS, consider vertices in the decreasing order of *f[u]*
- Output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

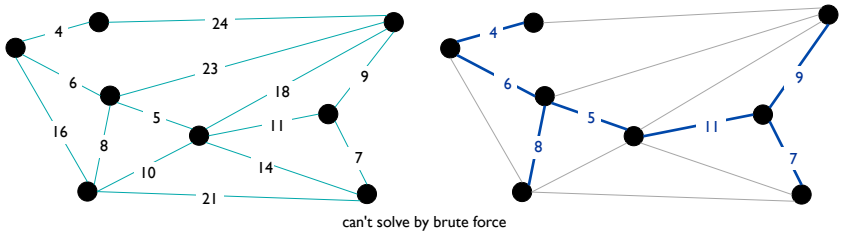


# Strongly connected components

- Property: Suppose you have two SCC's C and C'. If there is an edge between C and C', then vertex of C visited first has the highest finishing number: f(C) > f(C') suppose there is an edge u -> v from C to C'
- If DFS is started at C it visits all vertices in C and C' before it gets "stuck".

# Minimum Spanning Tree

• Minimum spanning tree. Given a connected graph G = (V, E) with real-valued edge weights $c_e$, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimized.




G = (V, E) T, $\Sigma_{e \in T} c_e = 50$

can't solve by brute force

• Cayley's Theorem. There are $n^{n-2}$ spanning trees of $K_n$ complete graph

# Applications

• MST is fundamental problem with diverse applications.

Network design.
telephone, electrical, hydraulic, TV cable, computer, road

Approximation algorithms for NP-hard problems.
traveling salesperson problem, Steiner tree

Indirect applications.
- max bottleneck paths
- LDPC codes for error correction
- image registration with Renyi entropy
- learning salient features for real-time face verification
- reducing data storage in sequencing amino acids in a protein
- model locality of particle interactions in turbulent fluid flows
- autoconfig protocol for Ethernet bridging to avoid cycles in a network

Cluster analysis.

# MST Algorithms

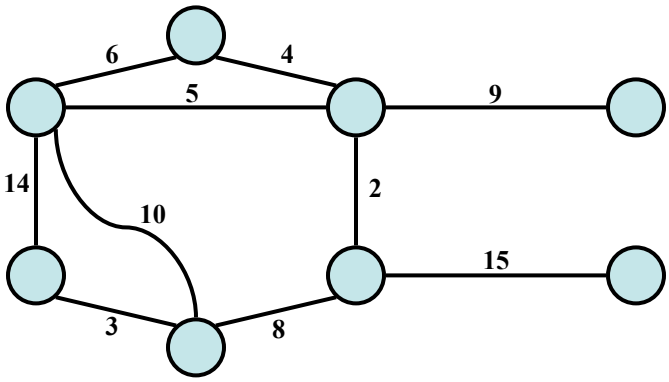
•Kruskal's algorithm. Start with $T = \phi$. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

•Reverse-Delete algorithm. Start with T = E. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.

•Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.

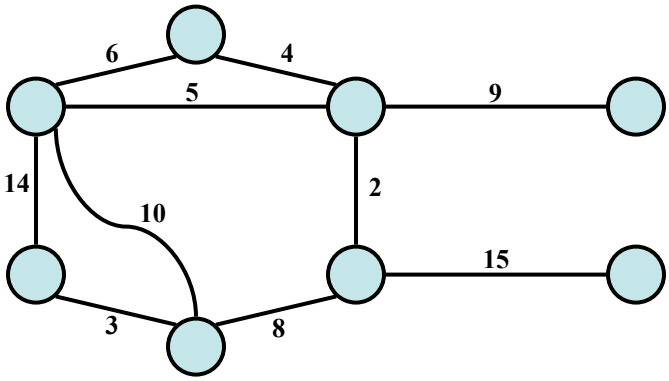
# Minimum Spanning Tree

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



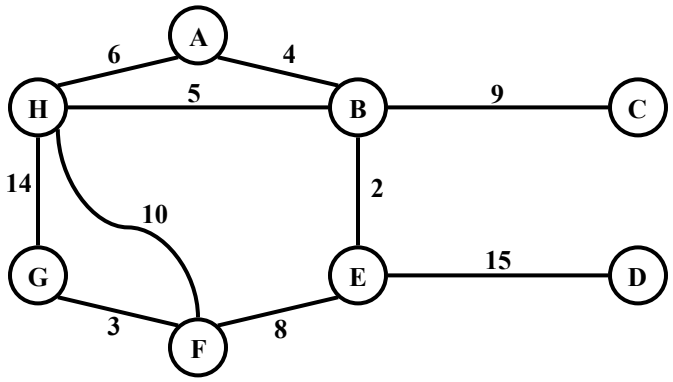

# Minimum Spanning Tree

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



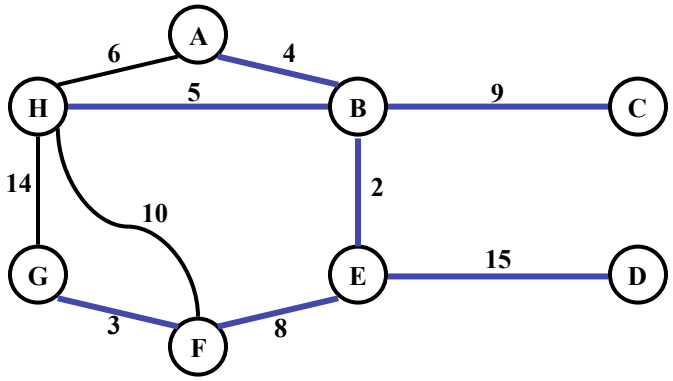
# Minimum Spanning Tree

- *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: 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 $G_2 = (V_2,E_2)$ (*Do $V_1$ and $V_2$ share vertices? Why?*)
  - 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)

# Minimum Spanning Tree

- Thm:
  Let T be MST of G, and let $A \subseteq T$ be subtree of T
  Let $(u,v)$ be min-weight edge connecting A to V-A
  Then $(u,v) \in T$

# Minimum Spanning Tree

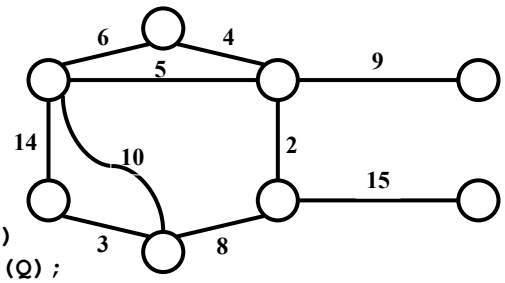
- Thm:
  Let T be MST of G, and let A $\subseteq$ T be subtree of T
  Let $(u,v)$ be min-weight edge connecting A to V-A
  Then $(u,v) \in$ T
- Proof: in book (see Thm 23.1)

# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

# Prim's Algorithm

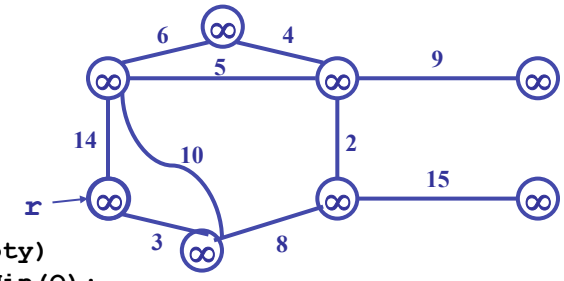
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```




Run on example graph

# Prim's Algorithm

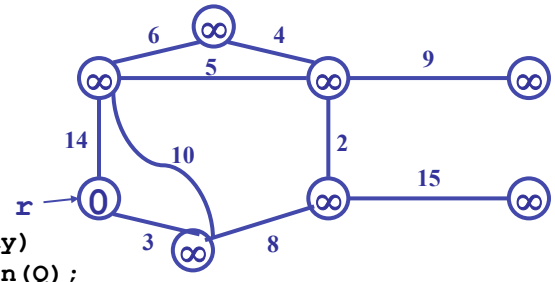
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```




Pick a start vertex r

# Prim's Algorithm

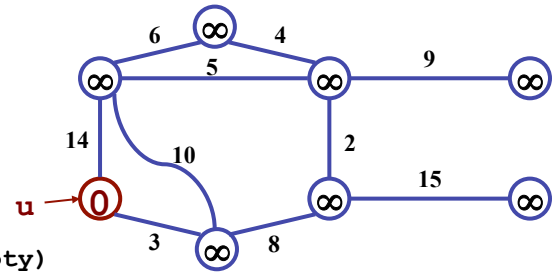
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```




Pick a start vertex r

# Prim's Algorithm

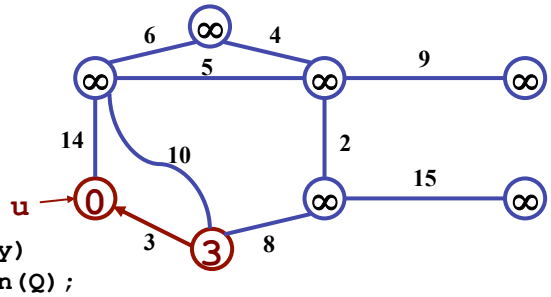
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```




Red vertices have been removed from Q

# Prim's Algorithm

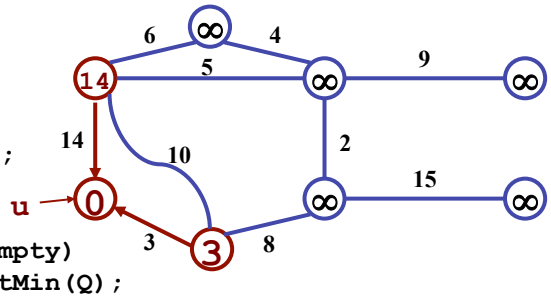
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```




Red arrows indicate parent pointers

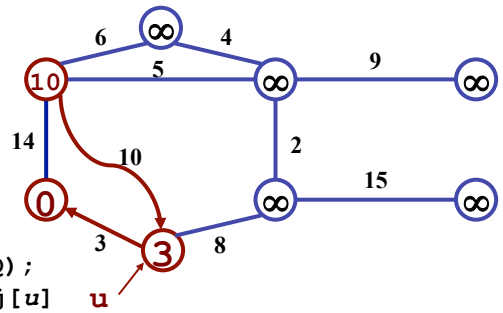
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



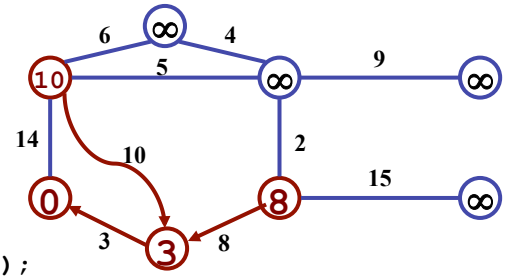
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



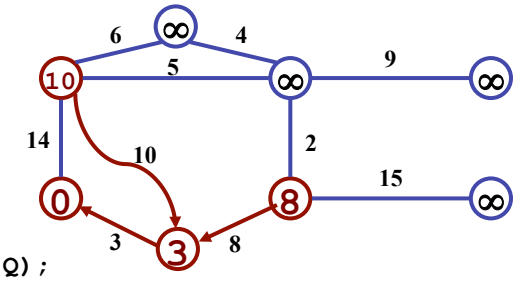

# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



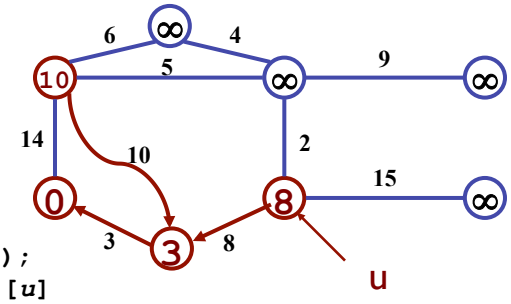
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



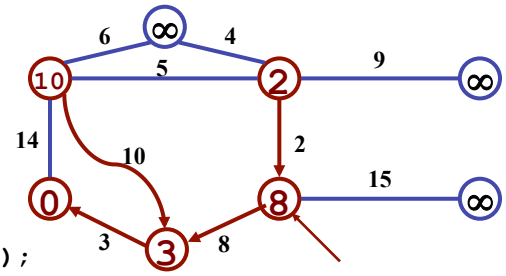

# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



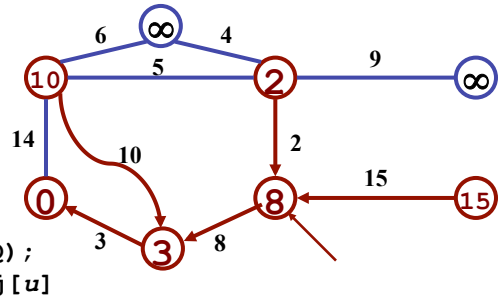
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



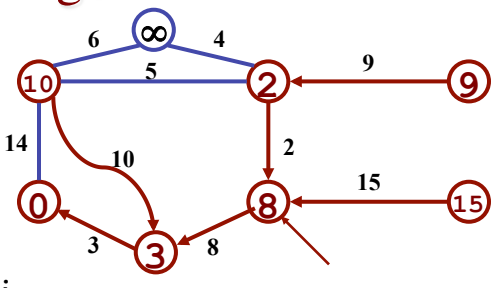

# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



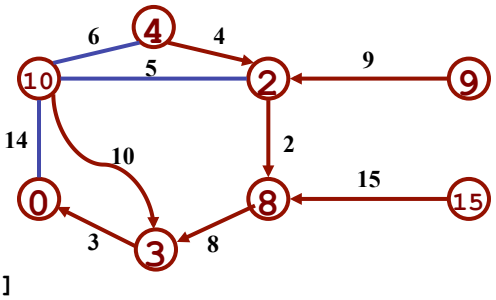
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



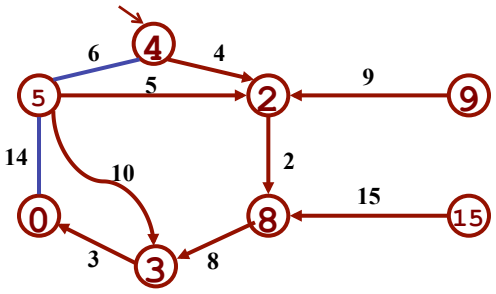

# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



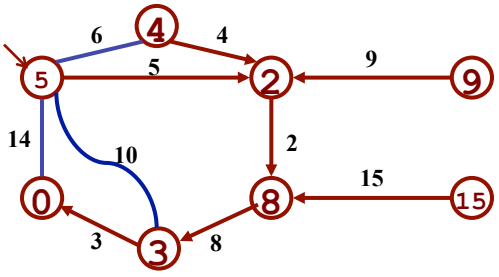
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



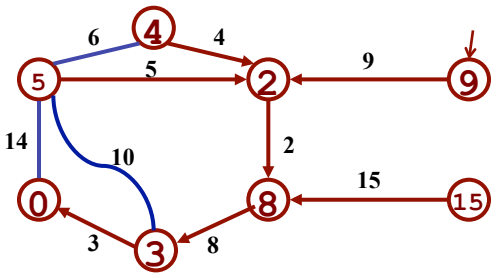

# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



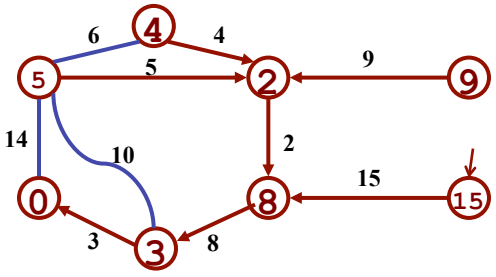
# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```




# Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



# Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                DecreaseKey(v, w(u,v));
```

# Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                DecreaseKey(v, w(u,v));
```

**How often is ExtractMin() called?**
**How often is DecreaseKey() called?**

ExtractMin total number of calls O(V log V)
DecreaseKey total number of calls O(E log V)

# Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

**What will be the running time?**
**A: Depends on queue**
**binary heap: O(E lg V)**
**Fibonacci heap: O(V lg V + E)**

ExtractMin total number of calls O(V log V)

DecreaseKey total number of calls O(E log V)

Total number of calls O(V logV +E logV) = O(E log V)

Think why we can combine things in the expression above

# Minimum Weight Spanning Tree
# Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

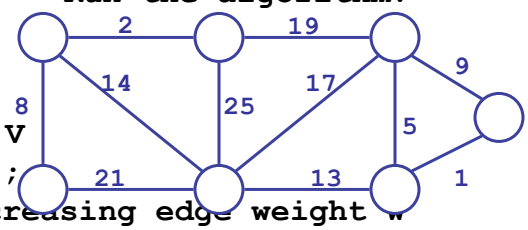
# Disjoint-Set Union Problem

- Want a data structure to support disjoint sets
  Collection of disjoint sets $S = \{S_i\}$, $S_i \cap S_j = \varnothing$
- Need to support following operations:
  MakeSet(x): $S = S \cup \{\{x\}\}$
  Union($S_i$, $S_j$): $S = S - \{S_i, S_j\} \cup \{S_i \cup S_j\}$
  FindSet(X): return $S_i \in S$ such that $x \in S_i$
- Before discussing implementation details, we look at example application: MSTs

# Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

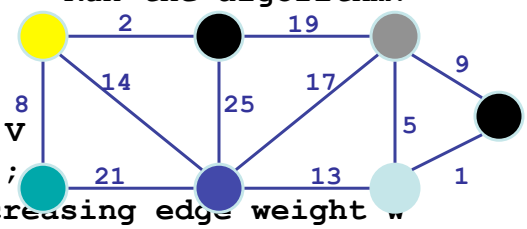
Run the algorithm:



# Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

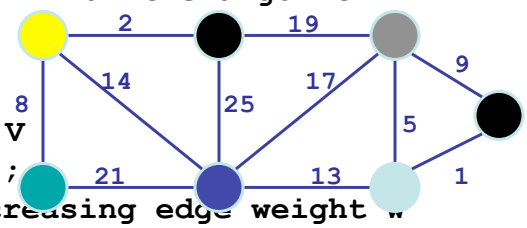
Run the algorithm:




# Kruskal's Algorithm

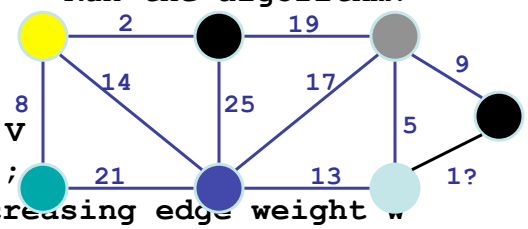
```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



# Kruskal's Algorithm

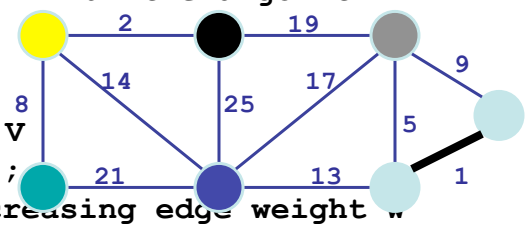
Run the algorithm:




```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

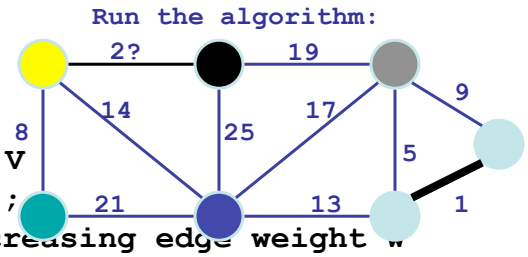
Run the algorithm:




```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

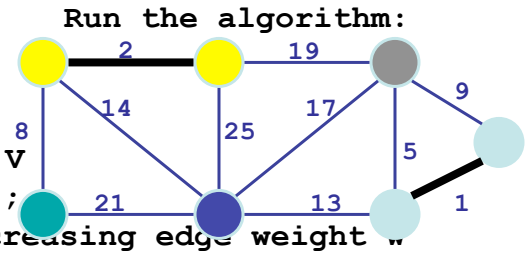
# Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



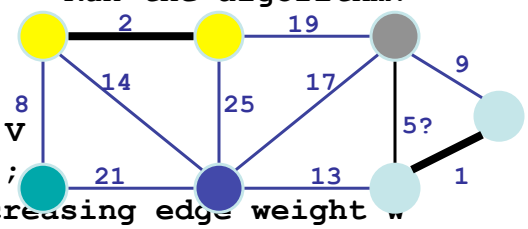

# Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```



# Kruskal's Algorithm

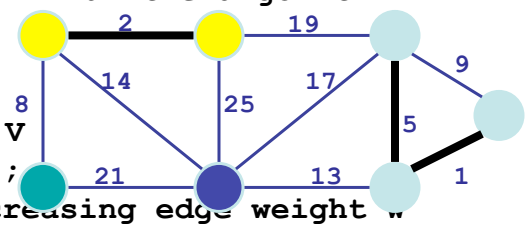
Run the algorithm:




```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

Run the algorithm:

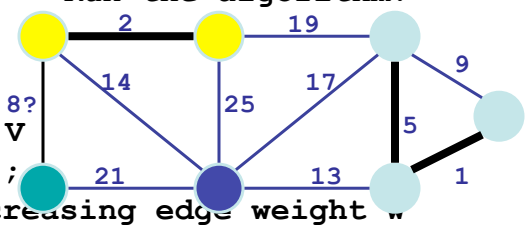



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

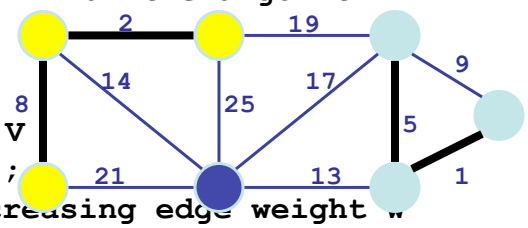
Run the algorithm:




# Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

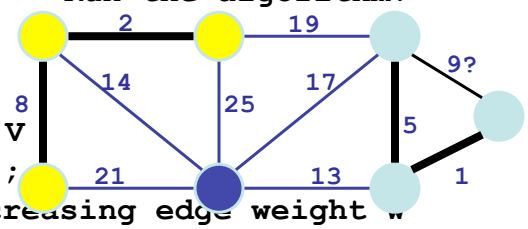
Run the algorithm:



# Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

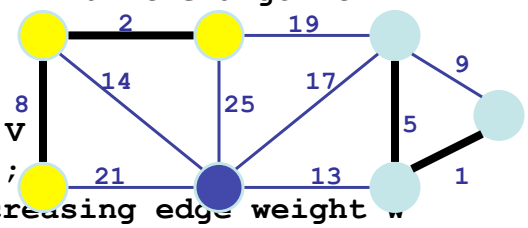
Run the algorithm:




# Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

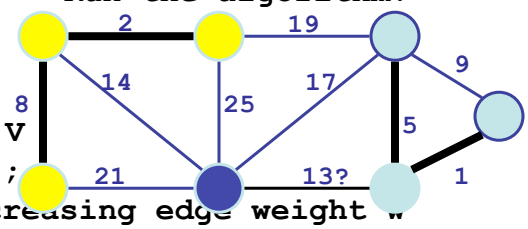
Run the algorithm:



# Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

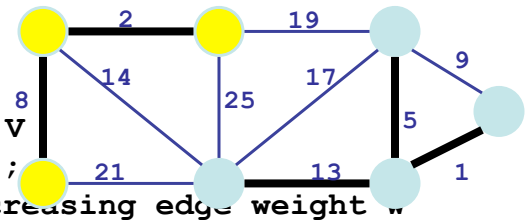
Run the algorithm:




# Kruskal's Algorithm

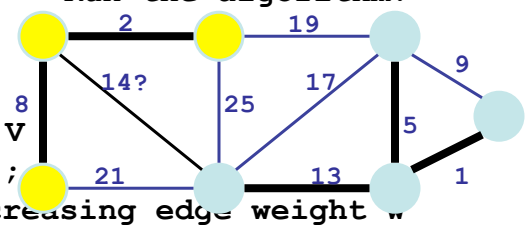
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



# Kruskal's Algorithm

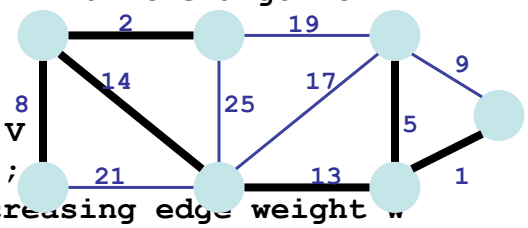
Run the algorithm:




```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

Run the algorithm:

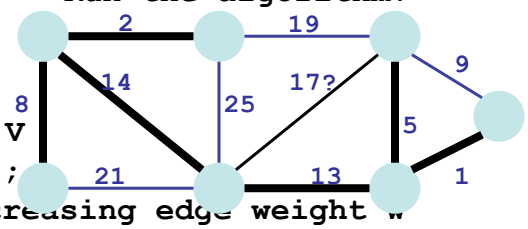



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

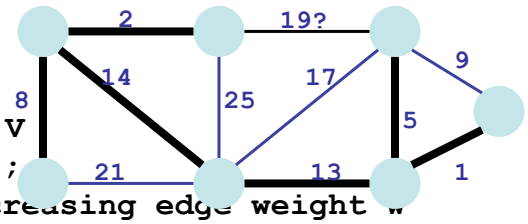
Run the algorithm:




# Kruskal's Algorithm

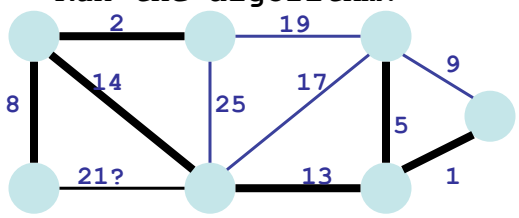
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



# Kruskal's Algorithm

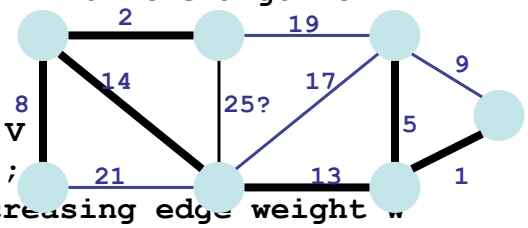
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

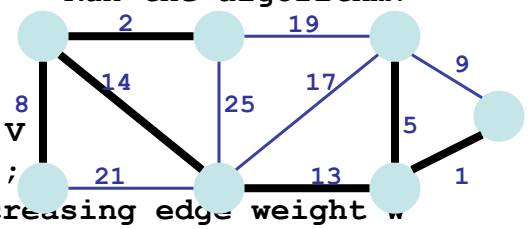
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

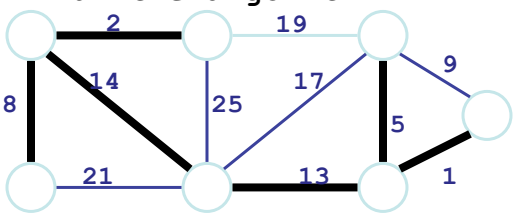
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

# Correctness Of Kruskal's Algorithm

- Sketch of a proof that this algorithm produces an MST for *T*:
  - Assume algorithm is wrong: result is not an MST
  - Then algorithm adds a wrong edge at some point
  - If it adds a wrong edge, there must be a lower weight edge (cut and paste argument)
  - But algorithm chooses lowest weight edge at each step -> Contradiction
- Again, important to be comfortable with cut and paste arguments

# Kruskal's Algorithm

**What will affect the running time?**

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

# Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

**What will affect the running time?**

**1 Sort**
**O(V) MakeSet() calls**
**O(E) FindSet() calls**
**O(V) Union() calls**
**(Exactly how many Union()s?)**

# Kruskal's Algorithm: Running Time

- To summarize:
  Sort edges: *O(E lg E)*
  *O(V)* MakeSet()'s
  *O(E)* FindSet()'s and Union()'s
- Upshot:
  Best disjoint-set union algorithm makes above
  3 operation stake $O((V+E)\cdot\alpha(V))$, $\alpha$ almost constant
  (slowly growing function of V)
  Since E >= V-1 then we have $O(E\cdot\alpha(V))$
  Also since $\alpha(V) = O(\lg V) = O(\lg E)$

  Overall thus *O(E lg E)*, almost linear w/o sorting

## Disjoint Sets (ch 21)

- In Kruskal's alg., Connected Components
- Need to do set membership and set union efficiently
- Typical operations on disjoint sets

```
member(a,s)
insert(a,s)
delete(a,s)
union(s1, s2, s3)
find(a)
make-set(x)
```

- Analysis in terms on n number of make-set operations
- And m total number of make-set, find, union (more details Later)

## Shortest Path Algorithms

## Single-Source Shortest Path

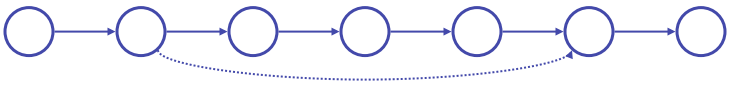
- Problem: given a weighted directed graph G, find the minimum-weight path from a given source vertex s to another vertex v

- "Shortest-path" = minimum weight
- Weight of path is sum of edges

- E.g., a road map: what is the shortest path from Faixfax to Washington DC?

## Shortest Path Properties

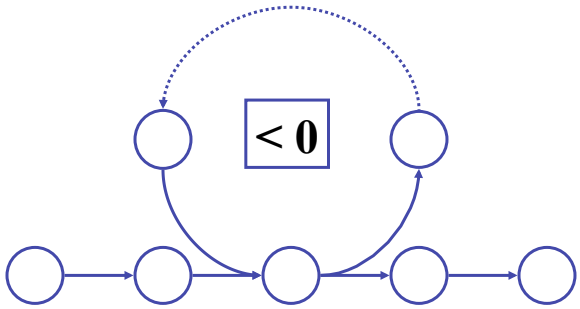
- Again, we have *optimal substructure*: the shortest path consists of shortest subpaths:



- Proof: suppose some subpath is not a shortest path
  There must then exist a shorter subpath
  Could substitute the shorter subpath for a shorter path
  but then overall path is not shortest path.
  Contradiction
- *Optimal substructure property* – hallmark of dynamic programming

# Shortest Path Properties

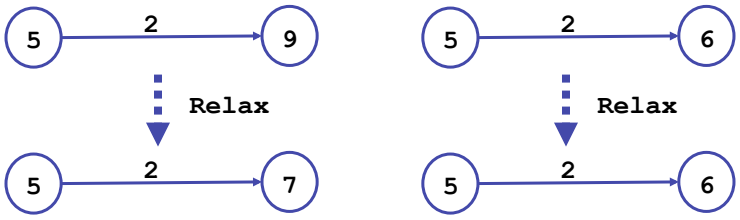
- In graphs with negative weight cycles, some shortest paths will not exist (*Why?*):




# Relaxation

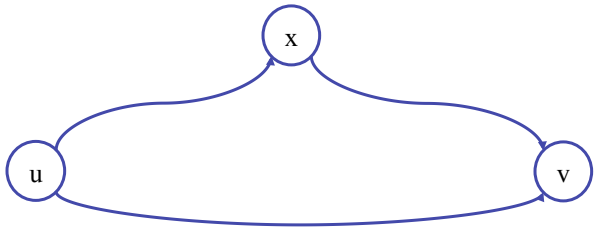
- A key technique in shortest path algorithms is *relaxation*
  Idea: for all $v$, maintain upper bound d[$v$] on δ(s,v)

```
Relax(u,v,w) {
    if (d[v] > d[u]+w) then d[v]=d[u]+w;
}
```



# Shortest Path Properties

- Define δ(u,v) to be the weight of the shortest path from u to v
- Shortest paths satisfy the *triangle inequality*: δ(u,v) ≤ δ(u,x) + δ(x,v)
- "Proof":




This path is no longer than any other path

# Dijkstra's Algorithm

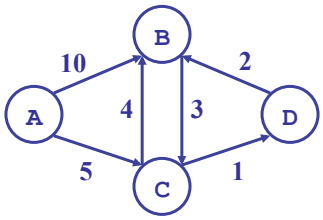
- If no negative edge weights, we can beat Bellman-Ford
- Similar to breadth-first search
- Grow a tree gradually, advancing from vertices taken from a queue
- Also similar to Prim's algorithm for MST
  Use a priority queue keyed on d[v]

# Dijkstra's Algorithm

```
Dijkstra(G)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S ∪ {u};
      for each v ∈ u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```

Relaxation Step

Note: this is really a call to Q->DecreaseKey()




Ex: run the algorithm

# Dijkstra's Algorithm

```
Dijkstra(G)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S ∪ {u};
      for each v ∈ u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```

How many times is ExtractMin() called?

How many times is DecraseKey() called?

What will be the total running time?

# Dijkstra's Algorithm

```
Dijkstra(G)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S ∪ {u};
      for each v ∈ u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```

How many times is ExtractMin() called?

How many times is DecraseKey() called?

A: O(E lg V) using binary heap for Q
Can acheive O(V lg V + E) with Fibonacci heaps

# Dijkstra's Algorithm

```
Dijkstra(G)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S ∪{u};
      for each v ∈ u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```

Correctness: we must show that when u is removed from Q, it has already converged