

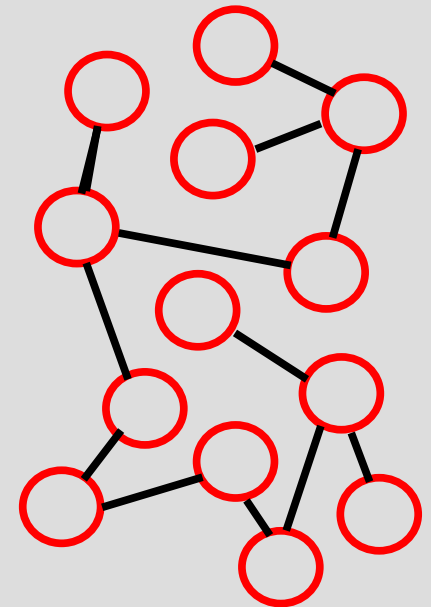
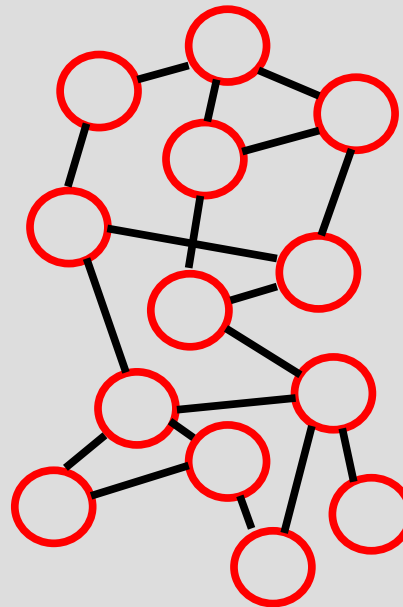
# Data Structures and Algorithms

## Week 10

1. Weighted Graphs
2. Minimum Spanning Trees
  - Greedy Choice Theorem
  - Kruskal's algorithm
  - Prim's algorithm
3. Shortest Paths
  - Dijkstra's algorithm
  - Bellman-Ford's algorithm

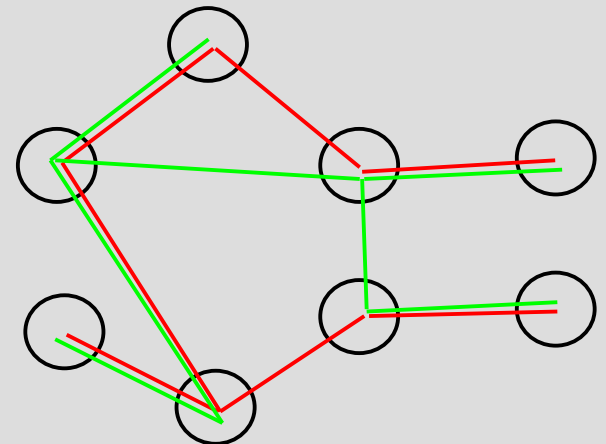
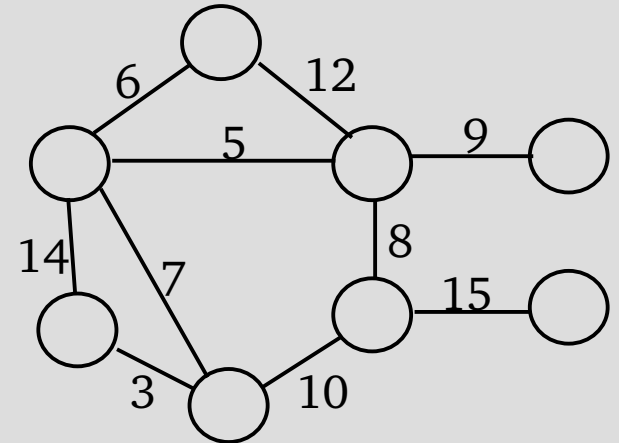
# Spanning Tree

- A **spanning tree** of  $G$  is a subgraph which
  - contains all vertices of  $G$
  - is a tree
- How many edges are there in a spanning tree, if there are  $V$  vertices?



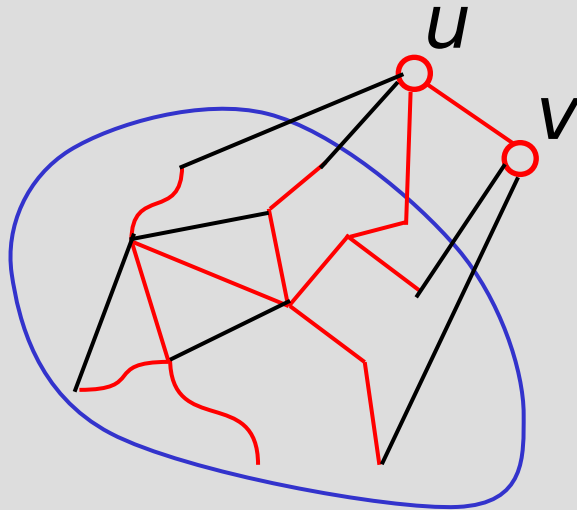
# Minimum Spanning Trees

- Undirected, connected graph  $G = (V, E)$
- **Weight** function  $W: E \rightarrow R$  (assigning cost or length or other values to edges)
- Spanning tree: tree that connects all vertexes
- **Minimum spanning tree (MST)**: spanning tree  $T$  that minimizes  $w(T) = \sum_{(u,v) \in T} w(u,v)$

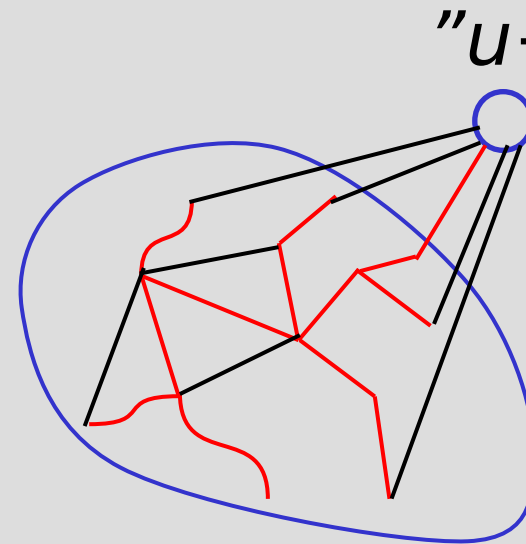


# Optimal Substructure

$$MST(G) = T$$



$$MST(G') = T - (u, v)$$



- Rationale:

- If  $G'$  would have a cheaper ST  $T'$ , then we would get a cheaper ST of  $G$ :  $T' + (u, v)$

# Idea for an Algorithm

- We have to make  $V-1$  choices (edges of the MST) to arrive at the optimization goal
- After each choice we have a sub-problem that is one vertex smaller than the original problem.
  - A dynamic programming algorithm would consider all possible choices (edges) at each vertex.
  - Goal: at each vertex cheaply determine an edge that definitely belongs to an MST

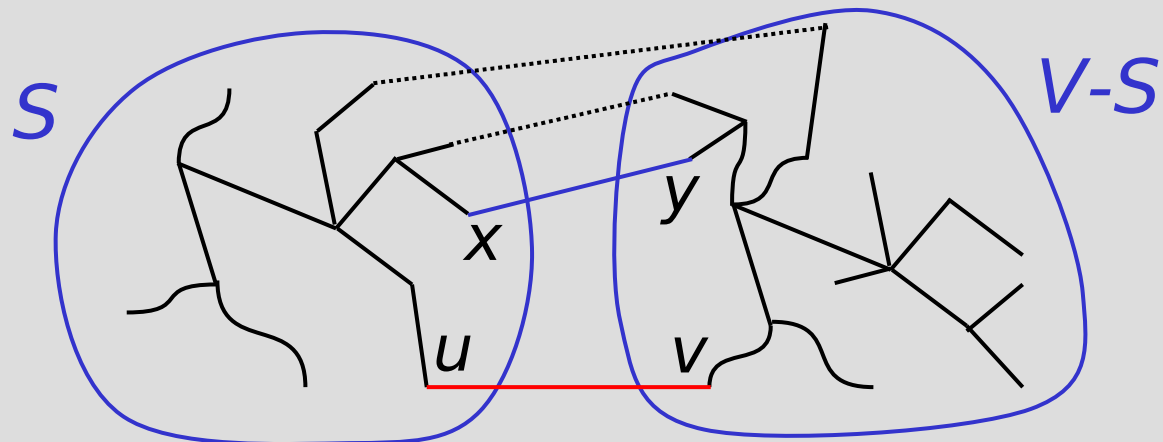
# Greedy Choice/1

- Greedy choice property: locally optimal (greedy) choice yields a globally optimal solution.
- Theorem
  - Let  $G=(V, E)$  and  $S \subseteq V$
  - $S$  is a **cut** of  $G$  (it splits  $G$  into parts  $S$  and  $V-S$ )
  - $(u,v)$  is a **light** edge if it is a *min*-weight edge of  $G$  that connects  $S$  and  $V-S$
  - Then  $(u,v)$  belongs to a MST  $T$  of  $G$

# Greedy Choice/2

6

- Proof
  - Suppose  $(u,v)$  is light but  $(u,v) \notin$  any MST
  - look at path from  $u$  to  $v$  in some MST  $T$
  - Let  $(x, y)$  be the first edge on a path from  $u$  to  $v$  in  $T$  that crosses from  $S$  to  $V-S$ . Swap  $(x, y)$  with  $(u,v)$  in  $T$ .
  - this improves cost of  $T \rightarrow$  contradiction ( $T$  is supposed to be an MST)



# Generic MST Algorithm

```
Generic-MST(G, w)
1 A :=  $\emptyset$  // Contains edges that belong to a MST
2 while A does not form a spanning tree do
3     Find an edge (u,v) that is safe for A
4     A := A  $\cup$  {(u,v)}
5 return A
```

A *safe edge* is an edge that does not destroy A's property.

```
MoreSpecific-MST(G, w)
1 A :=  $\emptyset$  // Contains edges that belong to a MST
2 while A does not form a spanning tree do
3.1 Make a cut (S, V-S) of G that respects A
3.2 Take the min-weight edge (u,v) connecting S to V-S
4 A := A  $\cup$  {(u,v)}
5 return A
```

# Prim-Jarnik Algorithm/1

- Vertex based algorithm
- Grows a single MST  $T$  one vertex at a time
- The set  $A$  covers the portion of  $T$  that was already computed
- Annotate all vertices  $v$  outside of the set  $A$  with  $v.\text{key}$  the minimum weight of an edge that connects  $v$  to a vertex in  $A$  ( $v.\text{key} = \infty$  if no such edge exists)

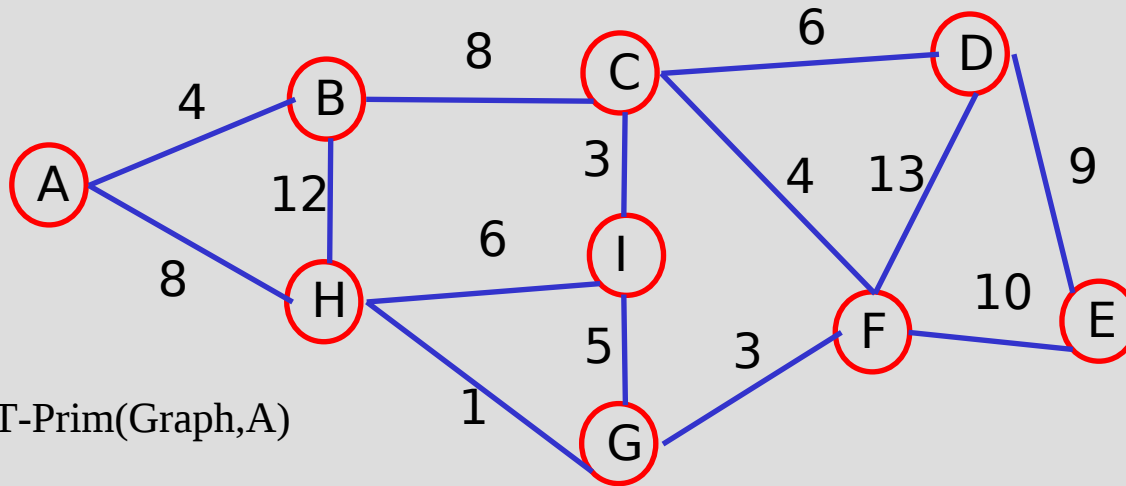
# Prim-Jarnik Algorithm/2

**MST-Prim**( $G, s$ )

```
01 for each vertex  $u \in G.V$ 
02    $u.key := \infty$ 
03    $u.pred := NIL$ 
04  $s.key := 0$ 
05 init( $Q, G.V$ ) //  $Q$  is a priority queue
06 while not isEmpty( $Q$ )
07    $u := extractMin(Q)$  // add  $u$  to  $T$ 
08   for each  $v \in u.adj$  do
09     if  $v \in Q$  and  $w(u, v) < v.key$  then
10        $v.key := w(u, v)$ 
11       modifyKey( $Q, v$ )
12        $v.pred := u$ 
```

updating  
keys

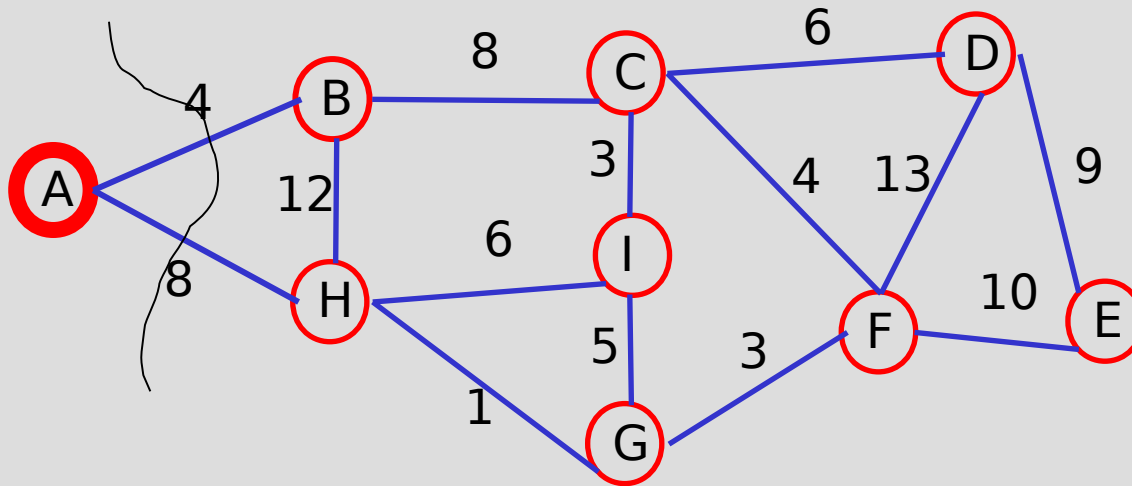
# Prim-Jarnik's Example/1



$$A = \{\}$$

$$Q = A\text{-NIL}/0, B\text{-NIL}/\infty, C\text{-NIL}/\infty, D\text{-NIL}/\infty, E\text{-NIL}/\infty, \\ F\text{-NIL}/\infty, G\text{-NIL}/\infty, H\text{-NIL}/\infty, I\text{-NIL}/\infty$$

# Prim-Jarnik's Example/2

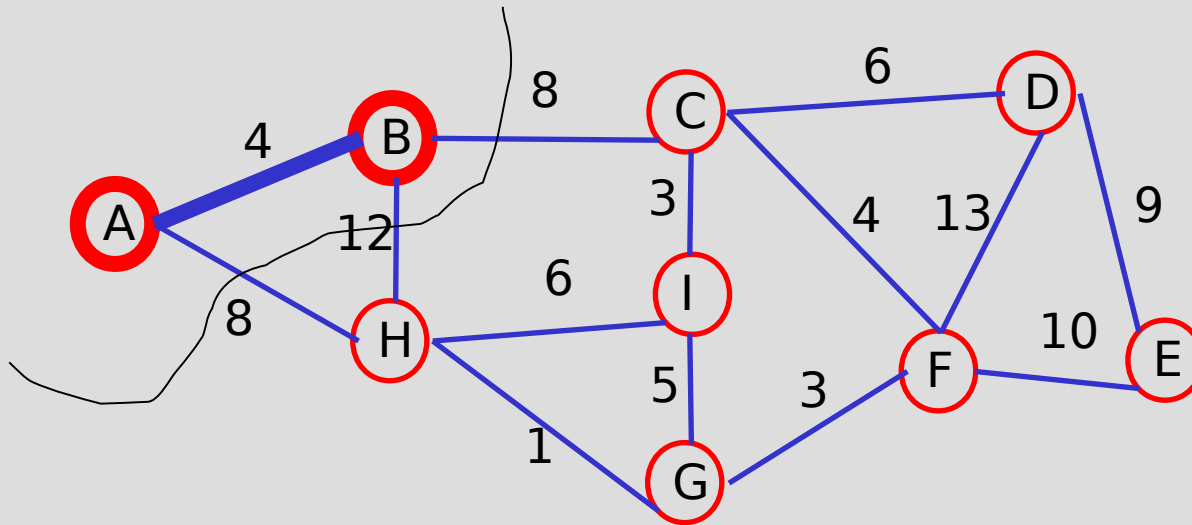


$A = A\text{-NIL}/0$

$Q = B\text{-}A/4, H\text{-}A/8, C\text{-NIL}/\infty, D\text{-NIL}/\infty, E\text{-NIL}/\infty,$   
 $F\text{-NIL}/\infty, G\text{-NIL}/\infty, I\text{-NIL}/\infty$

# Prim-Jarnik's Example/3

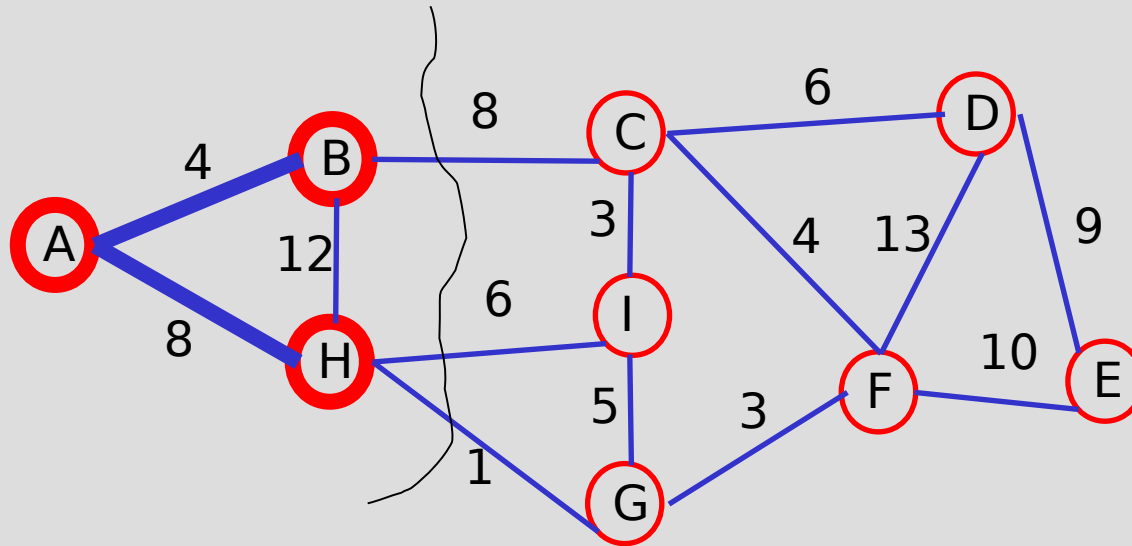
8



$A = A\text{-NIL}/0, B\text{-}A/4$

$Q = H\text{-}A/8, C\text{-}B/8, D\text{-NIL}/\infty, E\text{-NIL}/\infty,$   
 $F\text{-NIL}/\infty, G\text{-NIL}/\infty, I\text{-NIL}/\infty$

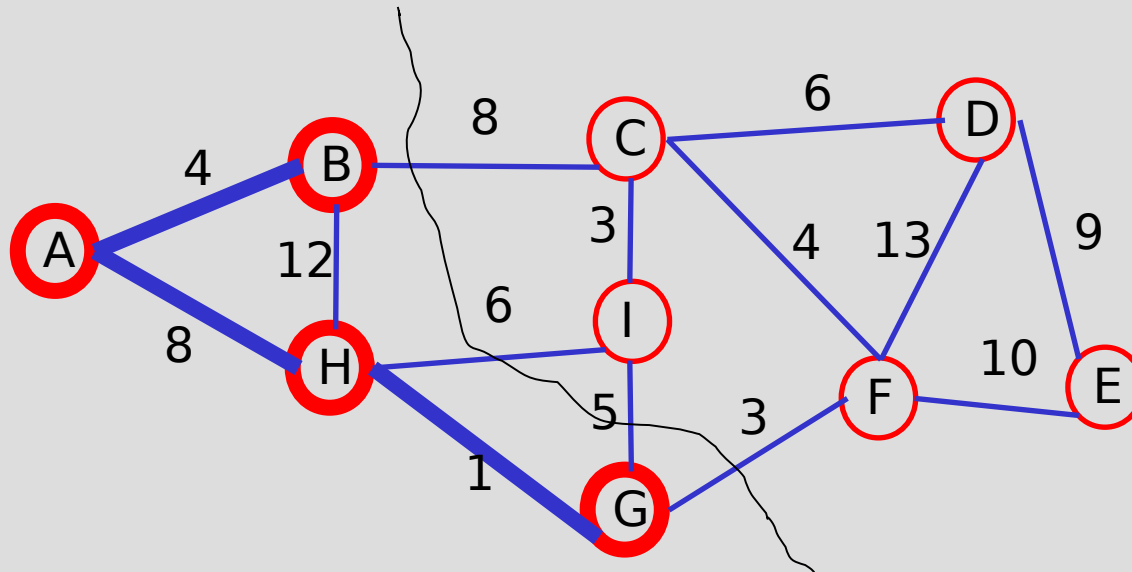
# Prim-Jarnik's Example/4



$A = A\text{-NIL}/0, B\text{-}A/4, H\text{-}A/8$

$Q = G\text{-}H/1, I\text{-}H/6, C\text{-}B/8, D\text{-NIL}/\infty, E\text{-NIL}/\infty, F\text{-NIL}/\infty$

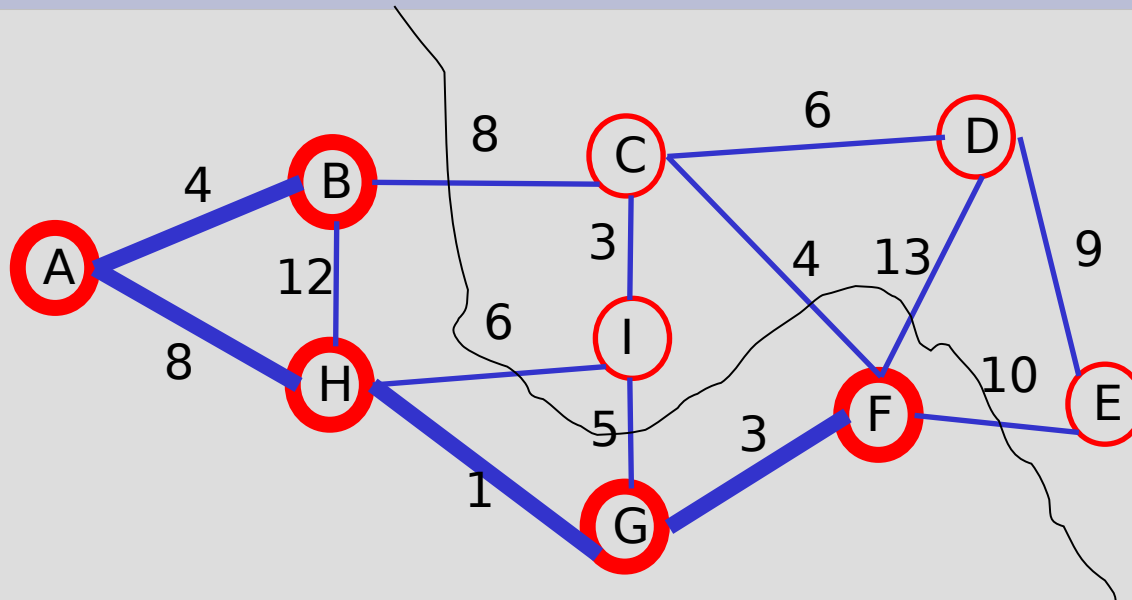
# Prim-Jarnik's Example/5



$A = A\text{-NIL}/0, B\text{-}A/4, H\text{-}A/8, G\text{-}H/1$

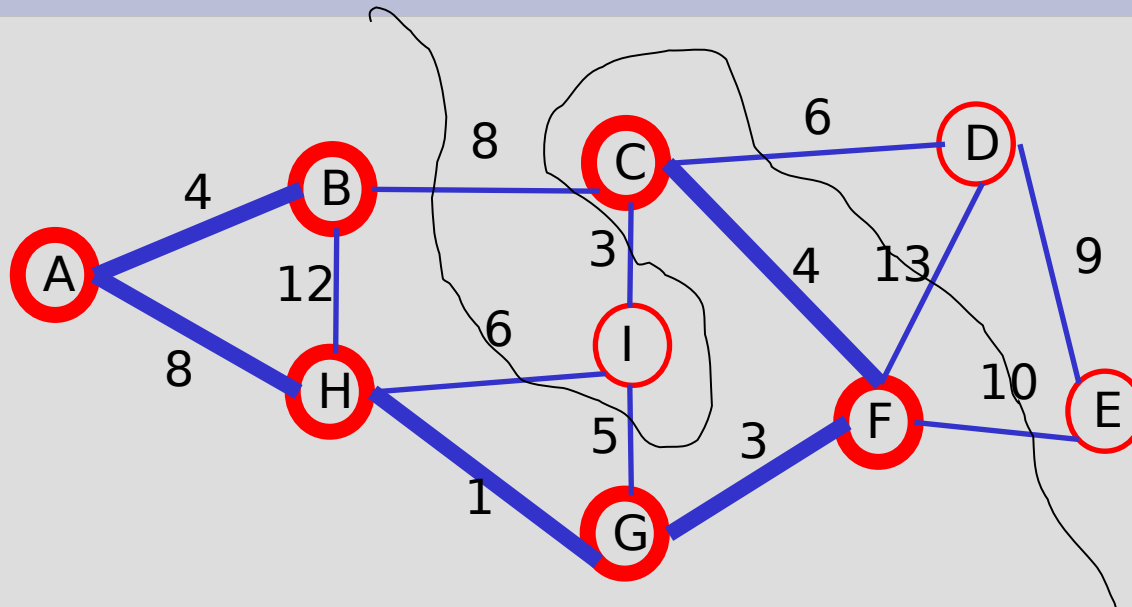
$Q = F\text{-}G/3, I\text{-}G/5, C\text{-}B/8, D\text{-}NIL/\infty, E\text{-}NIL/\infty$

# Prim-Jarnik's Example/6



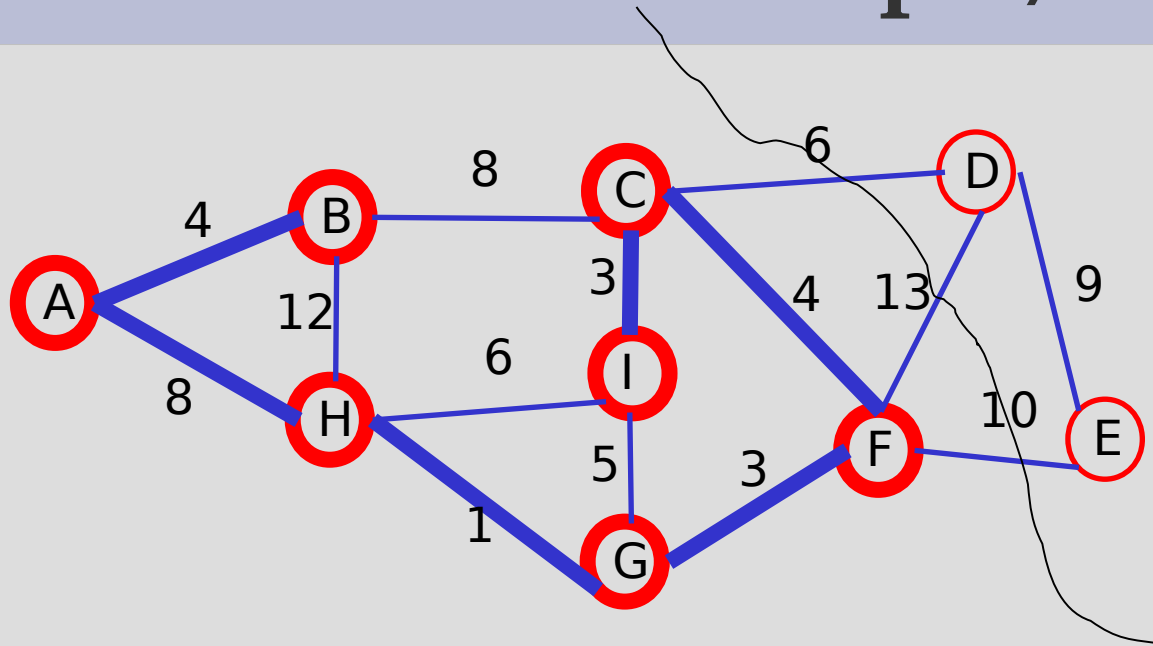
$A = A\text{-NIL}/0, B\text{-}A/4, H\text{-}A/8, G\text{-}H/1, F\text{-}G/3$   
 $Q = C\text{-}F/4, I\text{-}G/5, E\text{-}F/10, D\text{-}F/13$

# Prim-Jarnik's Example/7



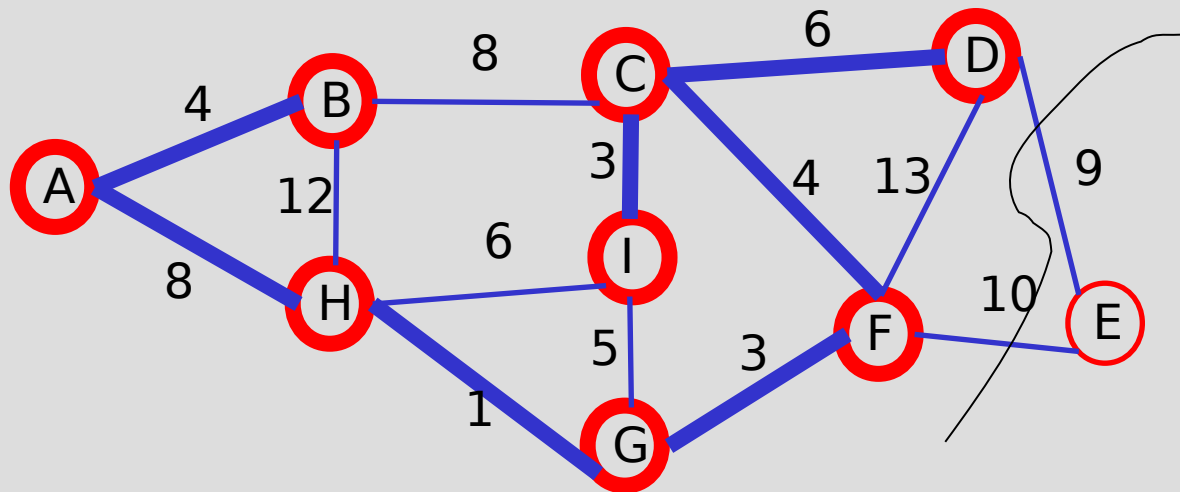
$A = A\text{-NIL}/0, B\text{-}A/4, H\text{-}A/8, G\text{-}H/1, F\text{-}G/3, C\text{-}F/4$   
 $Q = I\text{-}C/3, D\text{-}C/6, E\text{-}F/10$

# Prim-Jarnik's Example/8



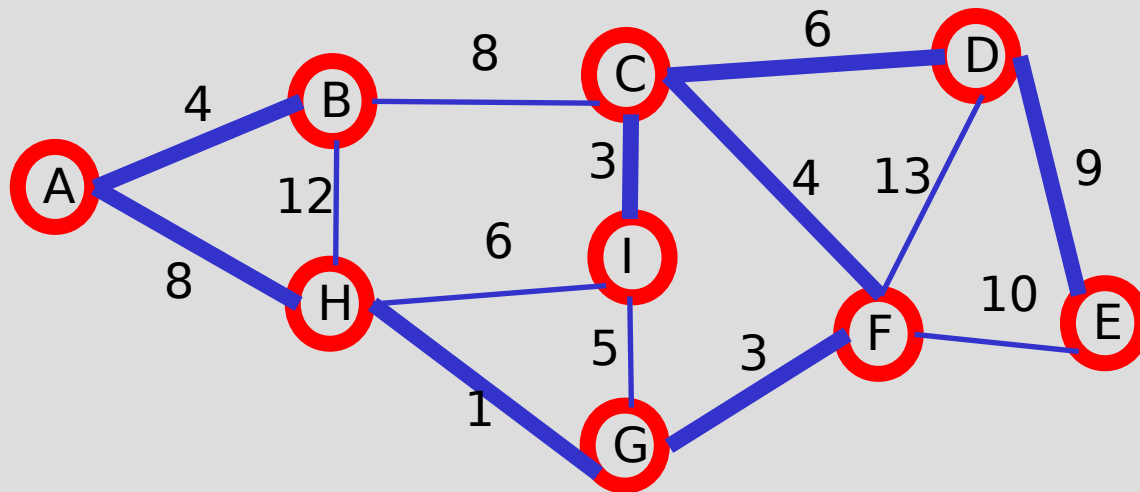
$A = A\text{-NIL}/0, B\text{-}A/4, H\text{-}A/8, G\text{-}H/1, F\text{-}G/3, C\text{-}F/4, I\text{-}C/3$   
 $Q = D\text{-}C/6, E\text{-}F/10$

# Prim-Jarnik's Example/9



A = A-NIL/0, B-A/4, H-A/8, G-H/1, F-G/3, C-F/4,  
I-C/3, D-C/6  
Q = E-D/9

# Prim-Jarnik's Example/10



$A = A\text{-NIL}/0, B\text{-}A/4, H\text{-}A/8, G\text{-}H/1, F\text{-}G/3, C\text{-}F/4,$   
 $I\text{-}C/3, D\text{-}C/6, E\text{-}D/9$

$Q = \{\}$

# Implementation Issues

9

## MST-Prim( $G, r$ )

```
01 for  $u \in G.V$  do  $u.key := \infty$ ;  $u.pred := NIL$ 
02  $r.key := 0$ 
03 init( $Q, G.V$ ) //  $Q$  is a min-priority queue
04 while not isEmpty( $Q$ ) do
05      $u := \text{extractMin}(Q)$  // add  $u$  to  $T$ 
06     for  $v \in u.adj$  do
07         if  $v \in Q$  and  $w(u, v) < v.key$  then
08              $v.key := w(u, v)$ 
09             modifyKey( $Q, v$ )
10              $v.pred := u$ 
```

# Priority Queues

- A priority queue maintains a set  $S$  of elements, each with an associated key value.
- We need PQ to support the following operations
  - **init**( $Q$ :PriorityQueue,  $S$ :VertexSet)
  - **extractMin**( $Q$ :PriorityQueue): *Vertex*
  - **modifyKey**( $Q$ :PriorityQueue,  $v$ :Vertex)
- To choose how to implement a PQ, we need to count how many times the operations are performed:
  - **init** is performed just once and runs in  $O(n)$

# Prim-Jarnik's Running Time

5

- Time =  $|V| * T(\text{extractMin}) + O(E) * T(\text{modifyKey})$

Q	T(extractMin)	T(modifyKey)	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\log V)$	$O(\log V)$	$O(E \log V)$

- $E \geq V-1, E < V^2, E = O(V^2)$
- Binary heap implementation:
  - Time =  $O(V \log V + E \log V) = O(V^2 \log V) = O(E \log V)$

# About Greedy Algorithms

- Greedy algorithms make a locally optimal choice (cheapest path, etc).
- In general, a locally optimal choice does not give a globally optimal solution.
- **Greedy** algorithms can be used to solve optimization problems, if:
  - There is an *optimal substructure*
  - We can prove that a *greedy choice* at each iteration leads to an optimal solution.

# Kruskal's Idea for MSTs

- Edge based algorithm
- Add edges one at a time in increasing weight order.
- The algorithm maintains  $A$ : a **forest of trees**. An edge is accepted if it connects vertices of distinct trees (the cut respects  $A$ ).
- Initially trees consist of single nodes.

# Disjoint Sets

1  
2

- We need to maintain a disjoint partitioning of a set, i.e., a collection  $S$  of disjoint sets.

Operations:

- **addSingletonSet**( $S:\text{Set}, x:\text{Vertex}$ )
  - $S := S \cup \{\{x\}\}$
- **findSet**( $S:\text{Set}, x:\text{Vertex}$ ): *Set*
  - returns  $X \in S$  such that  $x \in X$
- **unionSets**( $S:\text{Set}, x:\text{Vertex}, y:\text{Vertex}$ )
  - $X := \text{findSet}(S:\text{Set}, x)$
  - $Y := \text{findSet}(S:\text{Set}, y)$
  - $S := S - \{X, Y\} \cup \{X \cup Y\}$

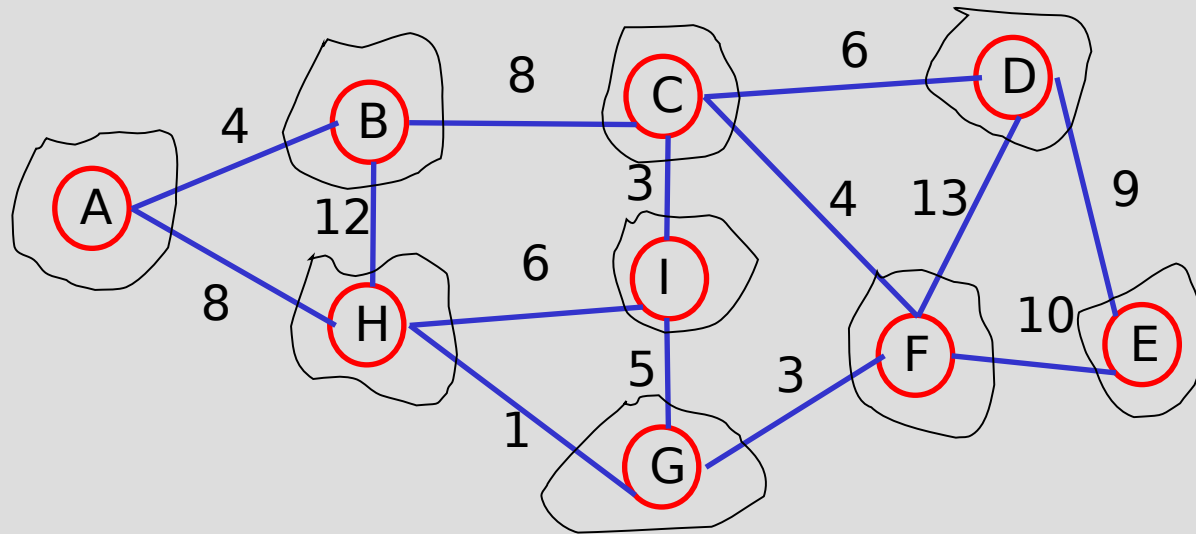
# Kruskal's Algorithm

- The algorithm keeps adding the cheapest edge that connects two trees of the forest

**MST-Kruskal(G)**

```
01 A :=  $\emptyset$ 
02 init(S) // Init disjoint-set
03 for v  $\in$  G.V do addSingletonSet(S, v)
05 sort edges of G.E by non-decreasing w(u, v)
06 for (u, v)  $\in$  G.E in sorted order do
07   if findSet(S, u)  $\neq$  findSet(S, v) then
08     A := A  $\cup$  {(u, v)}
09     unionSets(S, u, v)
10 return A
```

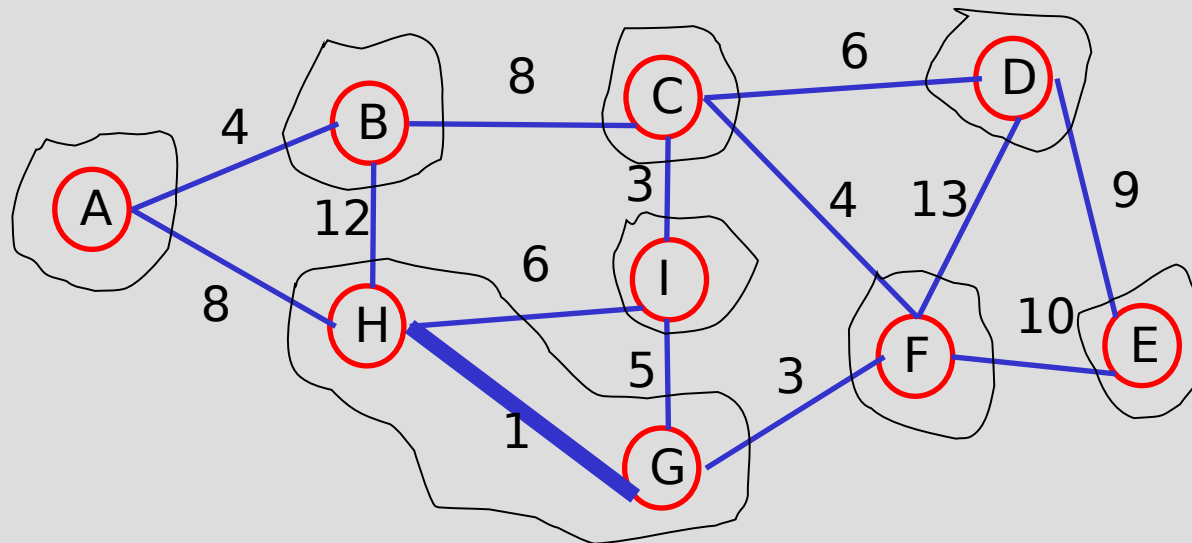
# Kruskal's Example/1



$S = \{ A B C D E F G H I \}$

$E' = \{ H G C I G F C F A B H I C D B C A H D E E F B H D F \}$

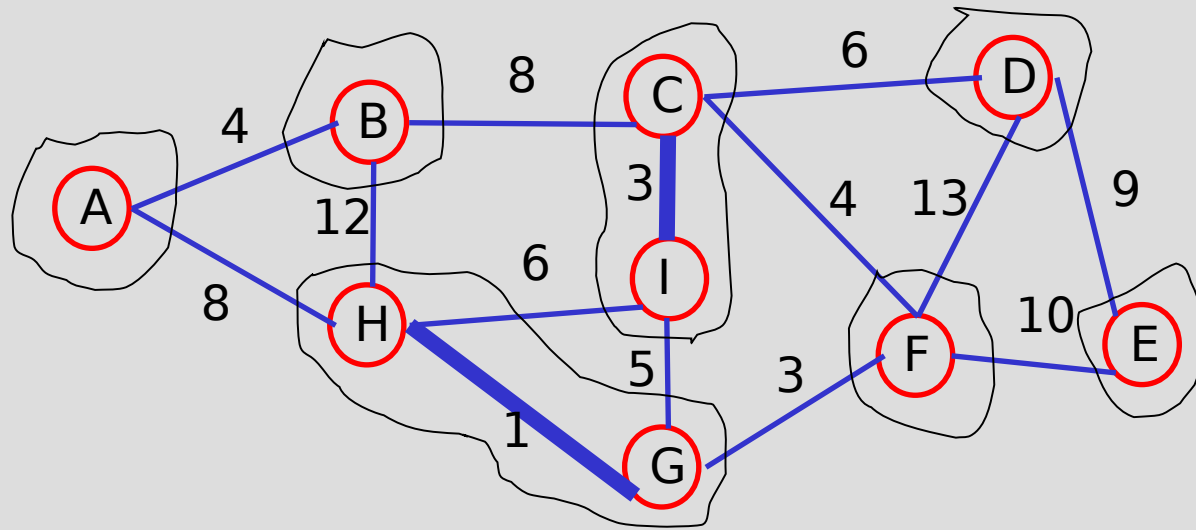
# Kruskal's Example/2



$S = \{ A B C D E F G H I \}$

$E' = \{ C I G F C F A B H I C D B C A H D E E F B H D F \}$

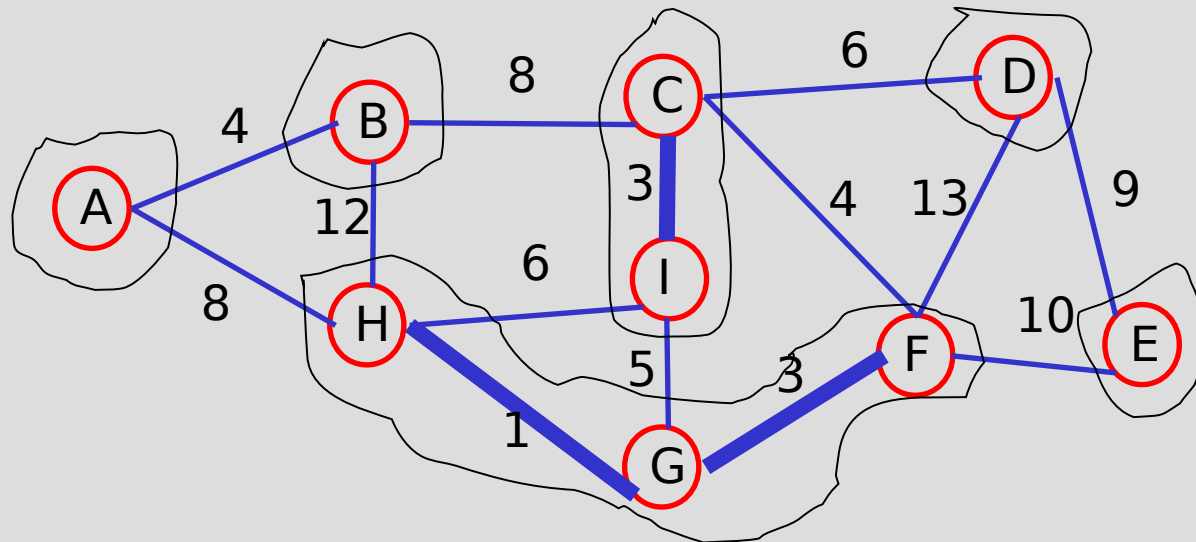
# Kruskal's Example/3



$S = \{ A B C I D E F G H \}$

$E' = \{ G F C F A B H I C D B C A H D E E F B H D F \}$

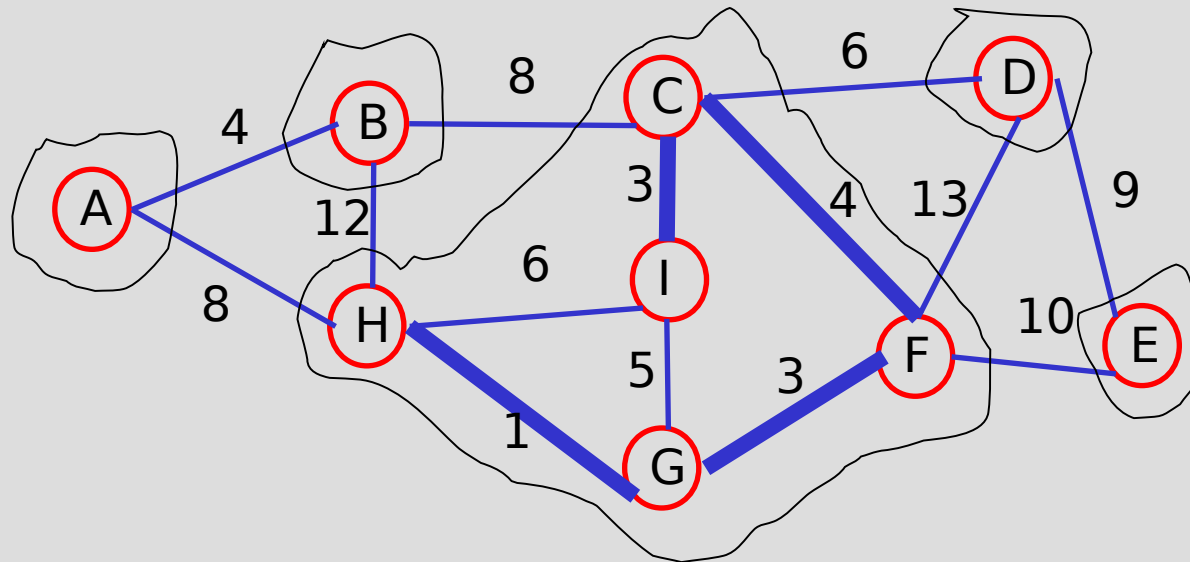
# Kruskal's Example/4



$S = \{ A B C I D E F G H \}$

$E' = \{ C F A B H I C D B C A H D E E F B H D F \}$

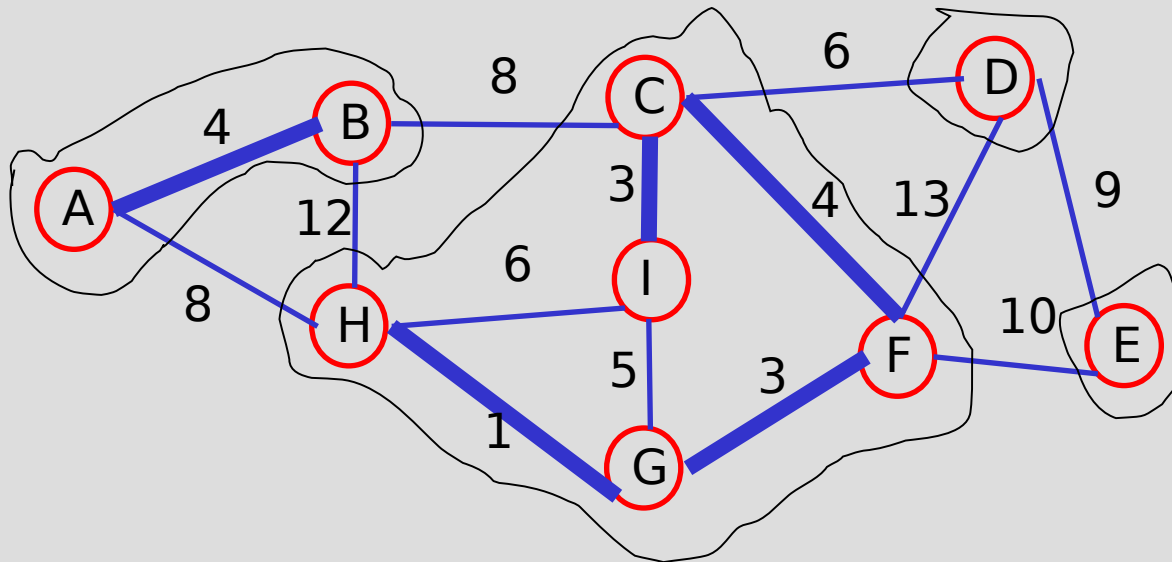
# Kruskal's Example/5



$S = \{ A B C F G H I D E \}$

$E' = \{ AB HI CD BC AH DE EF BH DF \}$

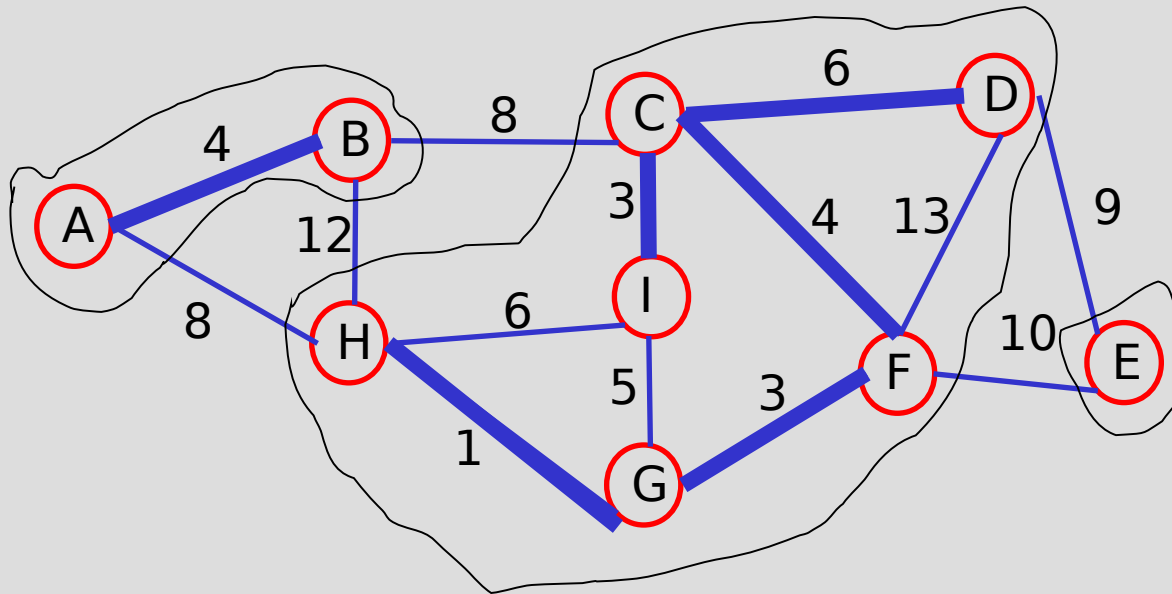
# Kruskal's Example/6



$S = \{ AB \text{ } CFGHI \text{ } D \text{ } E \}$

$E' = \{ HI \text{ } CD \text{ } BC \text{ } AH \text{ } DE \text{ } EF \text{ } BH \text{ } DF \}$

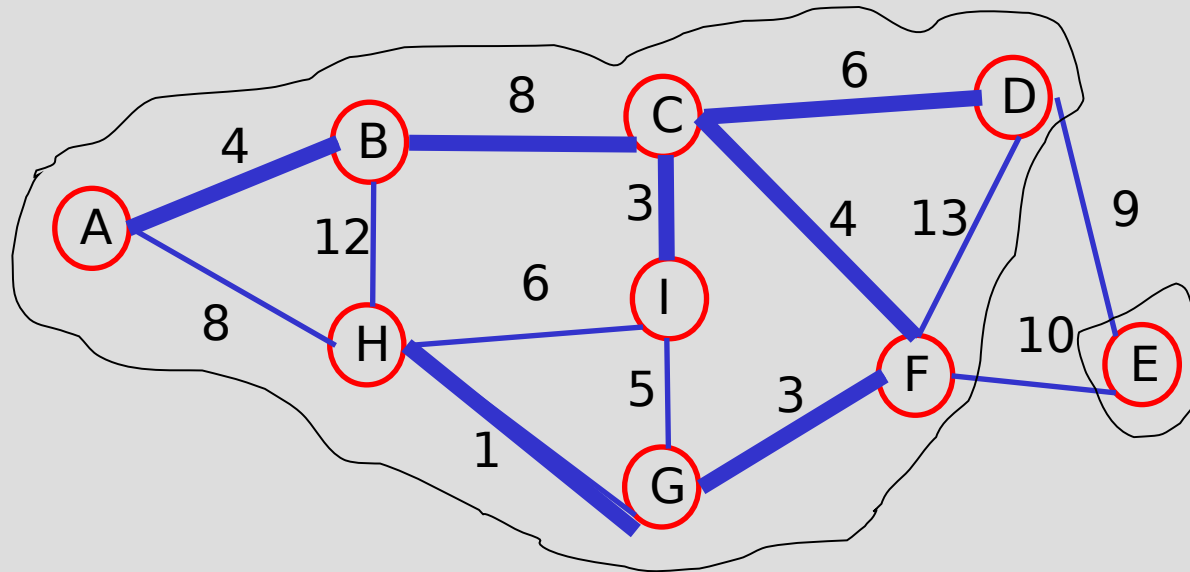
# Kruskal's Example/7



$S = \{ AB \ CD \ FG \ HI \ E \}$

$E' = \{ BC \ AH \ DE \ EF \ BH \ DF \}$

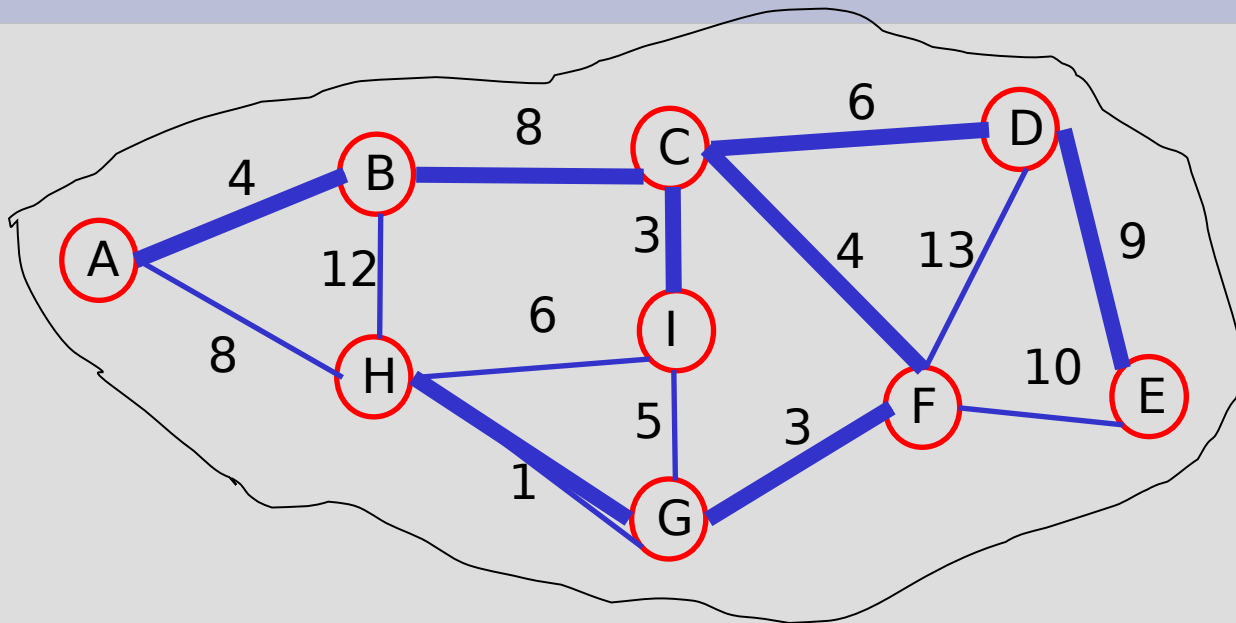
# Kruskal's Example/8



$S = \{ ABCDFGHI E \}$

$E' = \{ AH DE EF BH DF \}$

# Kruskal's Example/9



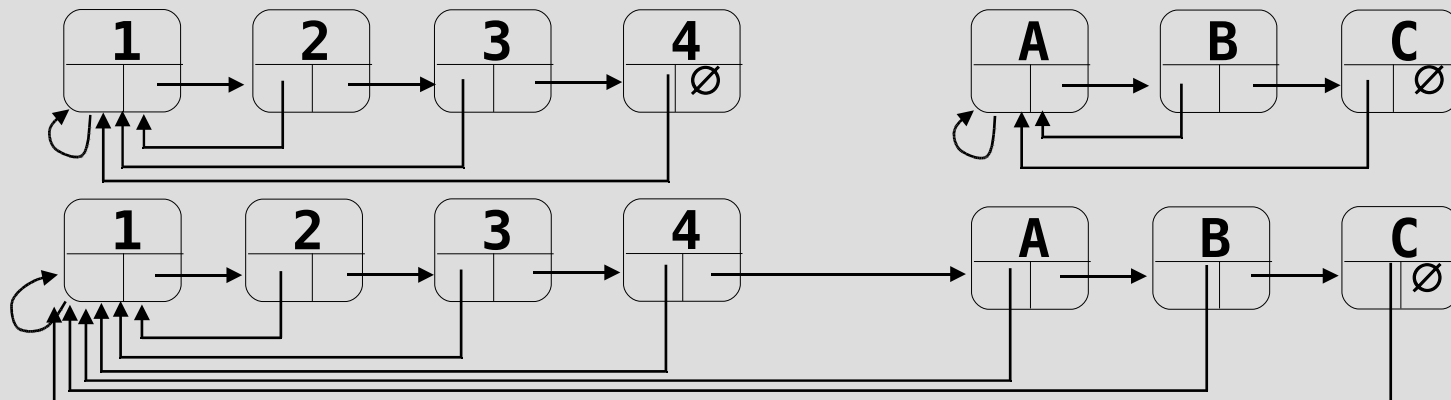
$S = \{ ABCDEFGHI \}$

$E' = \{ EF BH DF \}$

# Disjoint Sets as Lists

10

- Each set is a list of elements identified by the first element, all elements in the list point to the first element
- UnionSets: add a shorter list to a longer one,  $O(\min\{|C(u)|, |C(v)|\})$
- MakeSet/FindSet:  $O(1)$



# Kruskal Running Time

- Initialization  $O(V)$  time
- Sorting the edges  $\Theta(E \log E) = \Theta(E \log V)$
- $O(E)$  calls to FindSet
- Union costs
  - Let  $t(v)$  be the number of times  $v$  is moved to a new cluster
  - Each time a vertex is moved to a new cluster the size of the cluster containing the vertex at least doubles:  
 $t(v) \leq \log V$
  - Total time spent doing Union  $\sum_{v \in V} t(v) \leq |V| \log |V|$
- Total time:  $O(E \log V)$

# Shortest Path

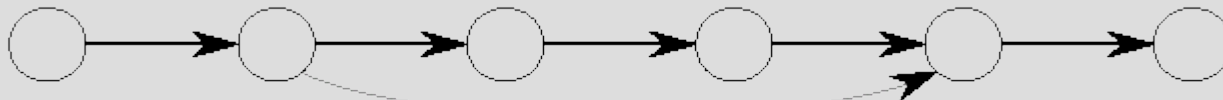
- Generalize directed graph to weighted setting
- Digraph  $G=(V,E)$  with weight function  $W: E \rightarrow R$  (assigning real values to edges)
- Weight of path  $p = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$  is
$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$
- Shortest path = a path of minimum weight (cost)
- Applications
  - static/dynamic network routing
  - robot motion planning
  - map/route generation in traffic

# Shortest-Path Problems

- Shortest-Path problems
  - **Single-source (single-destination)**. Find a shortest path from a given source (vertex  $s$ ) to each of the vertices.
  - **Single-pair**. Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
  - **All-pairs**. Find shortest-paths for every pair of vertices. Dynamic programming algorithm.
  - Unweighted shortest-paths – BFS.

# Optimal Substructure

- *Theorem*: subpaths of shortest paths are shortest paths
- *Proof*:
  - if some subpath were not the shortest path, one could substitute the shorter subpath and create a shorter total path



# Negative Weights and Cycles

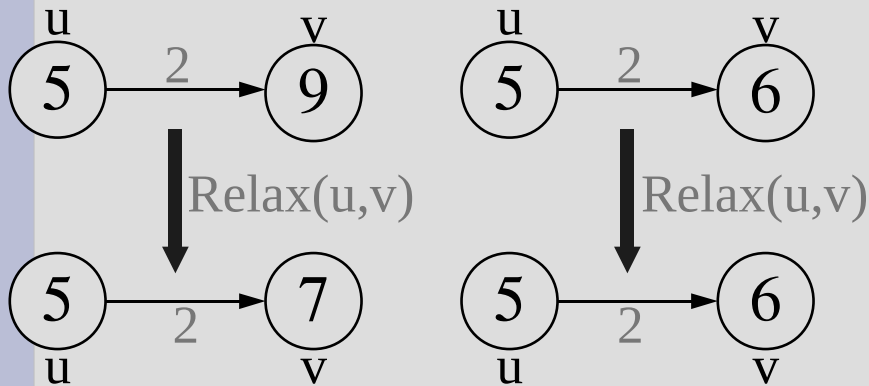
- Negative edges are OK, as long as there are no *negative weight cycles* (otherwise paths with arbitrary small “lengths” would be possible).
- Shortest-paths can have no cycles (otherwise we could improve them by removing cycles).
  - Any shortest-path in graph  $G$  can be no longer than  $n - 1$  edges, where  $n$  is the number of vertices.

# Shortest Path Tree

- The result of the algorithms is a *shortest path tree*. For each vertex  $v$ , it
  - records a shortest path from the start vertex  $s$  to  $v$ .
  - $v$ .**pred** is the predecessor of  $v$  in this shortest path
  - $v$ .**dist** is the shortest path length from  $s$  to  $v$

# Relaxation

- For each vertex  $v$  in the graph, we maintain  $v$ .**dist**, the estimate of the shortest path from  $s$ . *It is initialized to  $\infty$  at the start.*
- Relaxing an edge  $(u,v)$  means testing whether we can improve the shortest path to  $v$  found so far by going through  $u$ .



```
Relax (u, v, G)  
if v.dist > u.dist + w(u, v) then  
    v.dist := u.dist + w(u, v)  
    v.pred := u
```

# Dijkstra's Algorithm

- Non-negative edge weights
- Greedy, similar to Prim's algorithm for MST
- Like breadth-first search (if all weights = 1, one can simply use BFS)
- Use  $Q$ , a priority queue with keys  $v.\mathbf{dist}$  (BFS used FIFO queue, here we use a PQ, which is re-organized whenever some **dist** decreases)
- Basic idea
  - maintain a set  $S$  of solved vertices
  - at each step select "closest" vertex  $u$ , add it to  $S$ , and relax all edges from  $u$

# Dijkstra's Pseudo Code

Input: Graph  $G$ , start vertex  $s$

***Dijkstra***( $G, s$ )

```
01 for  $u \in G.V$ 
```

```
02      $u.dist := \infty$ 
```

```
03      $u.pred := NIL$ 
```

```
04  $s.dist := 0$ 
```

```
05 init( $Q, G.V$ ) // initialize priority queue  $Q$ 
```

```
06 while not isEmpty( $Q$ ) do
```

```
07      $u := \mathbf{extractMin}(Q)$ 
```

```
08     for  $v \in u.adj$  do
```

```
09         Relax( $u, v, G$ )
```

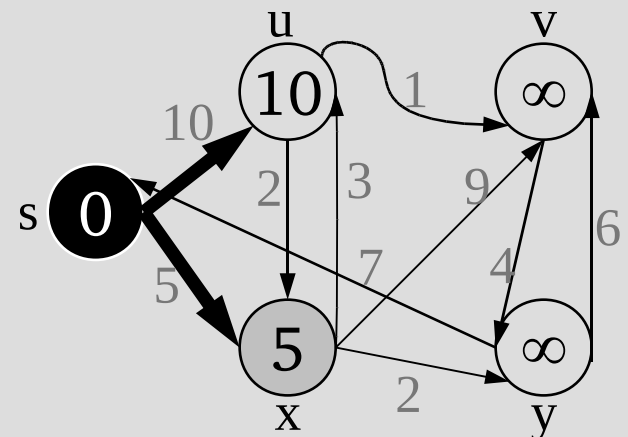
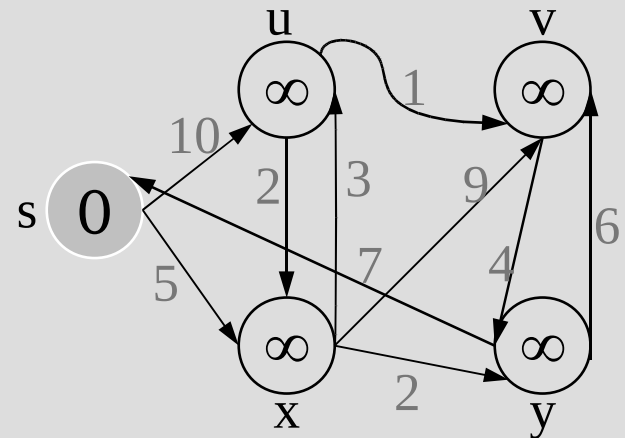
```
10         modifyKey( $Q, v$ )
```

initialize  
graph

relaxing  
edges

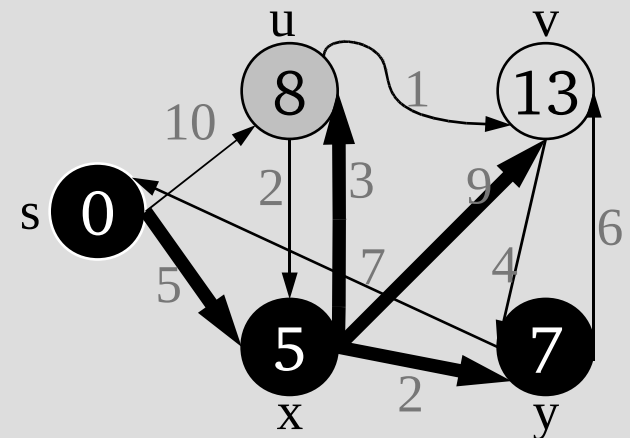
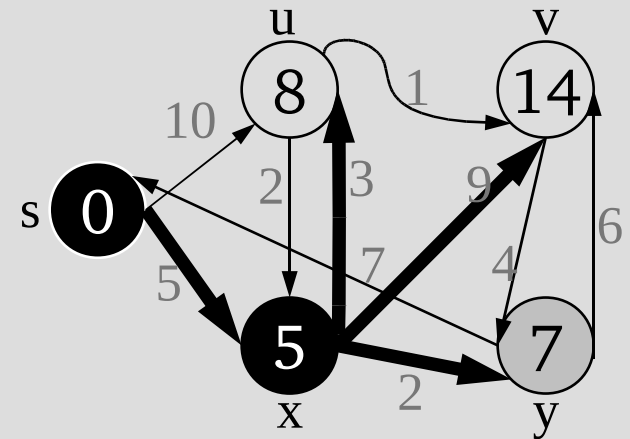
# Dijkstra's Example/1

```
Dijkstra(G, s)
01 for u ∈ G.V
02   u.dist := ∞
03   u.pred := NIL
04 s.dist := 0
05 init(Q, G.V)
06 while not isEmpty(Q) do
07   u := extractMin(Q)
08   for v ∈ u.adj do
09     Relax(u, v, G)
10     modifyKey(Q, v)
```



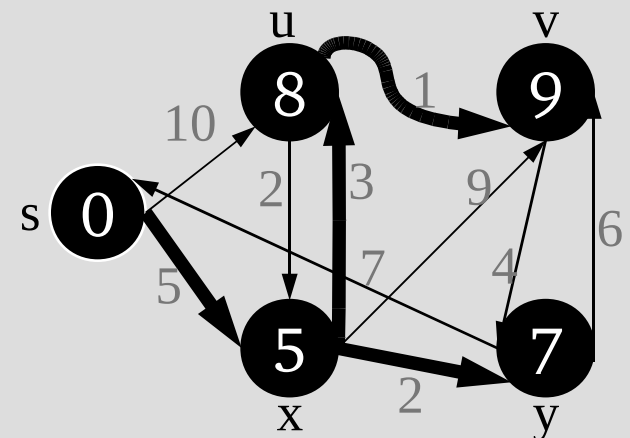
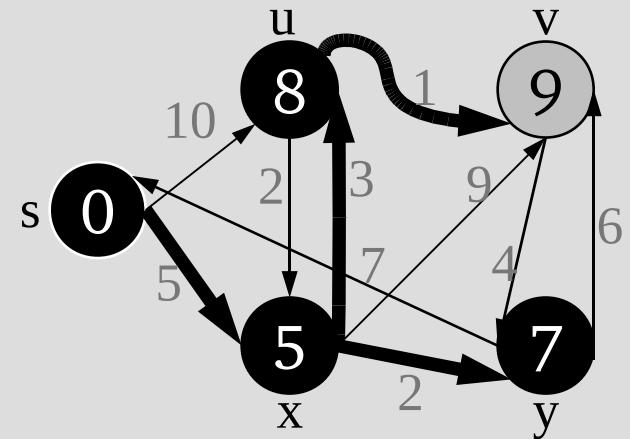
# Dijkstra's Example/2

```
Dijkstra(G, s)
01 for u ∈ G.V
02   u.dist := ∞
03   u.pred := NIL
04 s.dist := 0
05 init(Q, G.V)
06 while not isEmpty(Q) do
07   u := extractMin(Q)
08   for v ∈ u.adj do
09     Relax(u, v, G)
10     modifyKey(Q, v)
```



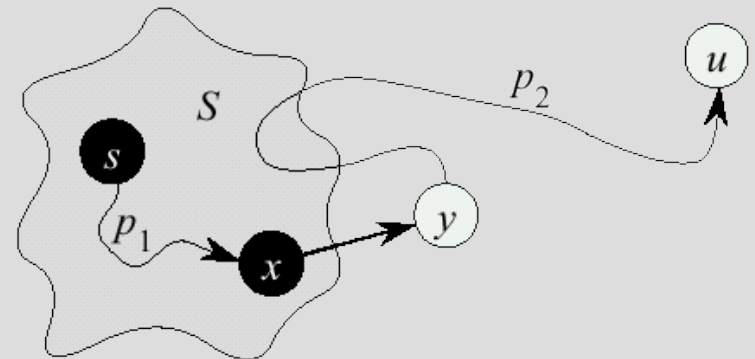
# Dijkstra's Example/3

```
Dijkstra(G, s)
01 for u ∈ G.V
02   u.dist := ∞
03   u.pred := NIL
04 s.dist := 0
05 init(Q, G.V)
06 while not isEmpty(Q) do
07   u := extractMin(Q)
08   for v ∈ u.adj do
09     Relax(u, v, G)
10     modifyKey(Q, v)
```



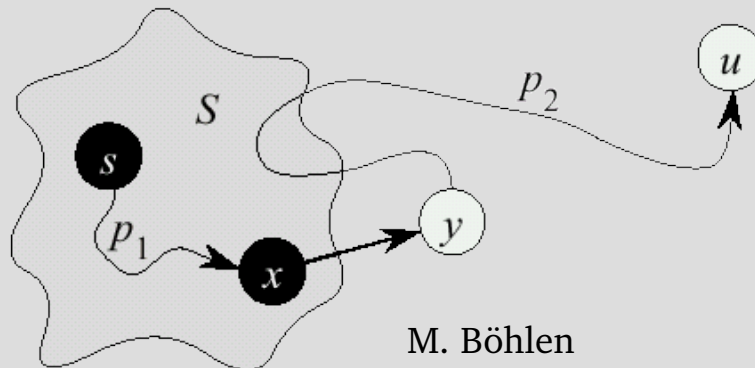
# Dijkstra's Correctness/1

- We prove that **whenever  $u$  is added to  $S$ ,  $u.\mathbf{dist} = \delta(s,u)$ , i.e.,  $\mathbf{dist}$  is minimum.**
- Proof (by contradiction)
  - Initially  $\forall v: v.\mathbf{dist} \geq \delta(s,v)$
  - Let  $u$  be the **first** vertex such that there is a shorter path than  $u.\mathbf{dist}$ , i.e.,  $u.\mathbf{dist} > \delta(s,u)$
  - We will show that this assumption leads to a contradiction



# Dijkstra Correctness/2

- Let  $y$  be the first vertex  $\in V-S$  on the actual shortest path from  $s$  to  $u$ , then it must be that  $y.\mathbf{dist} = \delta(s,y)$  because
  - $x.\mathbf{dist}$  is set correctly for  $y$ 's predecessor  $x \in S$  on the shortest path (by choice of  $u$  as the first vertex for which  $\mathbf{dist}$  is set incorrectly)
  - when the algorithm inserted  $x$  into  $S$ , it relaxed the edge  $(x,y)$ , setting  $y.\mathbf{dist}$  to the correct value



# Dijkstra Correctness/3

$$u.\mathbf{dist} > \delta(s,u)$$

$$= \delta(s,y) + \delta(y,u)$$

$$= y.\mathbf{dist} + \delta(y,u)$$

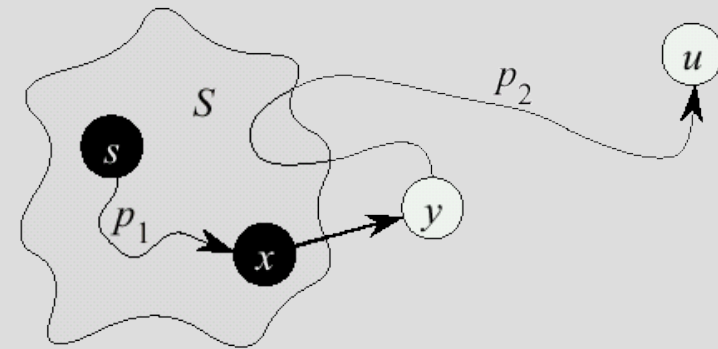
$$\geq y.\mathbf{dist}$$

*initial assumption*

*optimal substructure*

*correctness of  $y.\mathbf{dist}$*

*no negative weights*



- But  $u.\mathbf{dist} > y.\mathbf{dist} \Rightarrow$  algorithm would have chosen  $y$  (from the PQ) to process next, not  $u \Rightarrow$  contradiction
- Thus,  $u.\mathbf{dist} = \delta(s,u)$  at time of insertion of  $u$  into  $S$ , and Dijkstra's algorithm is correct

# Dijkstra's Running Time

- Extract-Min executed  $|V|$  time
- Decrease-Key executed  $|E|$  time
- Time =  $|V| T_{\text{Extract-Min}} + |E| T_{\text{Decrease-Key}}$
- $T$  depends on different Q implementations

Q	T(Extract-Min)	T(Decrease-Key)	Total
array	$O(V)$	$O(1)$	$O(V^2)$
heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$

# Bellman-Ford Algorithm/1

- Dijkstra's doesn't work when there are negative edges:
  - Intuition – we cannot be greedy anymore on the assumption that the lengths of paths will only increase in the future
- Bellman-Ford algorithm detects negative cycles (returns *false*) or returns the shortest path-tree

# Bellman-Ford Algorithm/2

## Bellman-Ford( $G, s$ )

```
01 for each vertex  $u \in G.V$ 
02    $u.dist := \infty$ 
03    $u.pred := NIL$ 
04  $s.dist := 0$ 
```

initialization

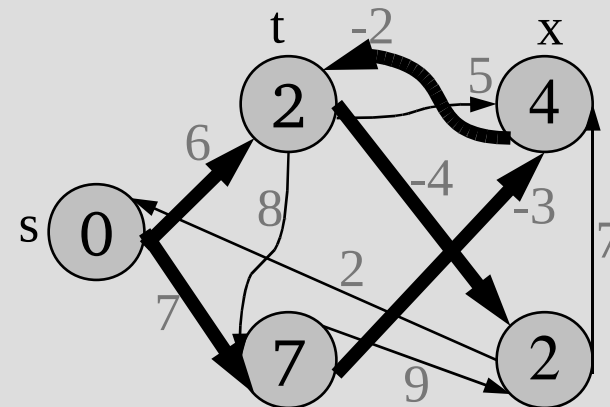
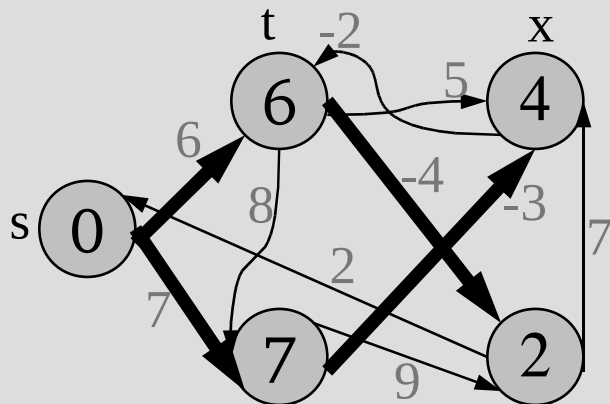
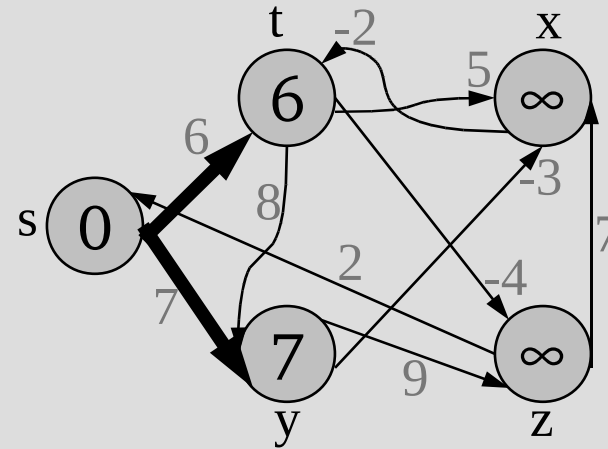
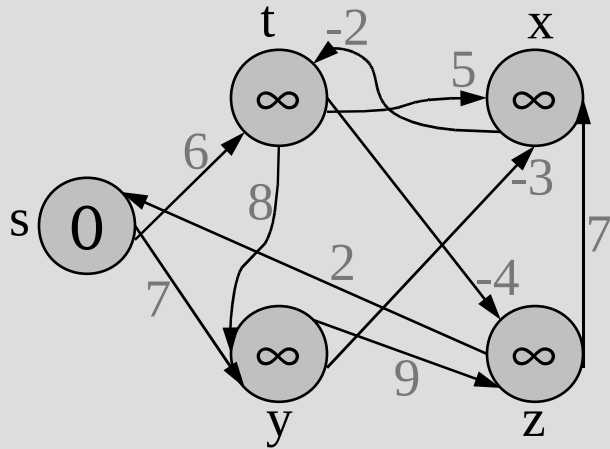
```
05 for  $i := 1$  to  $|G.V| - 1$  do
06   for each edge  $(u, v) \in G.E$  do
07     Relax  $(u, v, G)$ 
```

compute  
distances

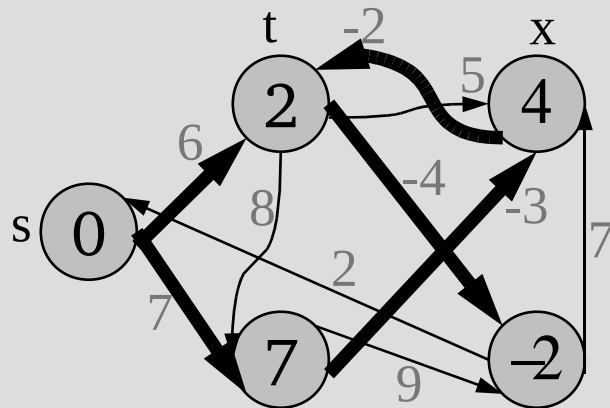
```
08 for each edge  $(u, v) \in G.E$  do
09   if  $v.dist > u.dist + w(u, v)$  then
10     return false
11 return true
```

check for  
negative cycles

# Bellman-Ford Example/1



# Bellman-Ford Example/2



- Bellman-Ford running time:
  - $|V| + (|V|-1) * |E| + |E| = O(VE)$

# Correctness of Bellman-Ford/1

- Let  $\delta_i(s,u)$  denote the length of path from  $s$  to  $u$ , that is shortest among all paths, that contain at most  $i$  edges
- Prove by induction that  $u.\mathbf{dist} = \delta_i(s,u)$  after the  $i^{\text{th}}$  iteration of Bellman-Ford
  - Base case ( $i=0$ ) trivial
  - Inductive step (say  $u.\mathbf{dist} = \delta_{i-1}(s,u)$ ):
    - Either  $\delta_i(s,u) = \delta_{i-1}(s,u)$
    - Or  $\delta_i(s,u) = \delta_{i-1}(s,z) + w(z,u)$
    - In an iteration we try to relax each edge  $((z,u)$  also), so we handle both cases, thus  $u.\mathbf{dist} = \delta_i(s,u)$

# Correctness of Bellman-Ford/2

- After  $n-1$  iterations,  $u.\mathbf{dist} = \delta_{n-1}(s,u)$ , for each vertex  $u$ .
- If there is some edge to relax in the graph, then there is a vertex  $u$ , such that  $\delta_n(s,u) < \delta_{n-1}(s,u)$ . But there are only  $n$  vertices in  $G$  – we have a cycle, and it is negative.
- Otherwise,  $u.\mathbf{dist} = \delta_{n-1}(s,u) = \delta(s,u)$ , for all  $u$ , since any shortest path will have at most  $n-1$  edges

# Shortest-Path in DAG's/1

- Finding shortest paths in DAG's is much easier, because it is easy to find an order in which to do relaxations – Topological sorting!

**DAG-Shortest-Paths**( $G, w, s$ )

```
01 for each vertex  $u \in G.V$ 
02    $u.dist := \infty$ 
03    $u.pred := NIL$ 
04  $s.dist := 0$ 
05 topologically sort  $G$ 
06 for each vertex  $u$  in topological order do
07   for each  $v \in u.adj$  do
08     Relax( $u, v, G$ )
```

# Shortest-Paths in DAG's/2

- Running time:
  - $\Theta(V+E)$  – only one relaxation for each edge,  $V$  times faster than Bellman-Ford

# Summary and Outlook

- Greedy algorithms
- MST: Kruskal
- MST: Prim
- Shortest path: Dijkstra
- Shortest path: Bellman-Ford
  
- Next: Introduction to complexity