

# CSE214 – Analysis of Algorithms

PhD Furkan Gözükar, Toros University

[https://github.com/FurkanGozukara/CSE214\\_2018](https://github.com/FurkanGozukara/CSE214_2018)

## Lecture 9

### Greedy Algorithms

*Based on Cevdet David Matuszek's  
Lecture Notes - Director of MCIT*

---



# Optimization problems

---

- An **optimization problem** is one in which you want to find, not just *a* solution, but the *best* solution
- A “greedy algorithm” sometimes works well for optimization problems
- A **greedy algorithm** works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum



# Example: Counting money

---

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm would do this would be:  
**At each step, take the largest possible bill or coin that does not overshoot**
  - Example: To make \$6.39, you can choose:
    - a \$5 bill
    - a \$1 bill, to make \$6
    - a 25¢ coin, to make \$6.25
    - A 10¢ coin, to make \$6.35
    - four 1¢ coins, to make \$6.39
- For US money, the greedy algorithm always gives the optimum solution



# A failure of the greedy algorithm

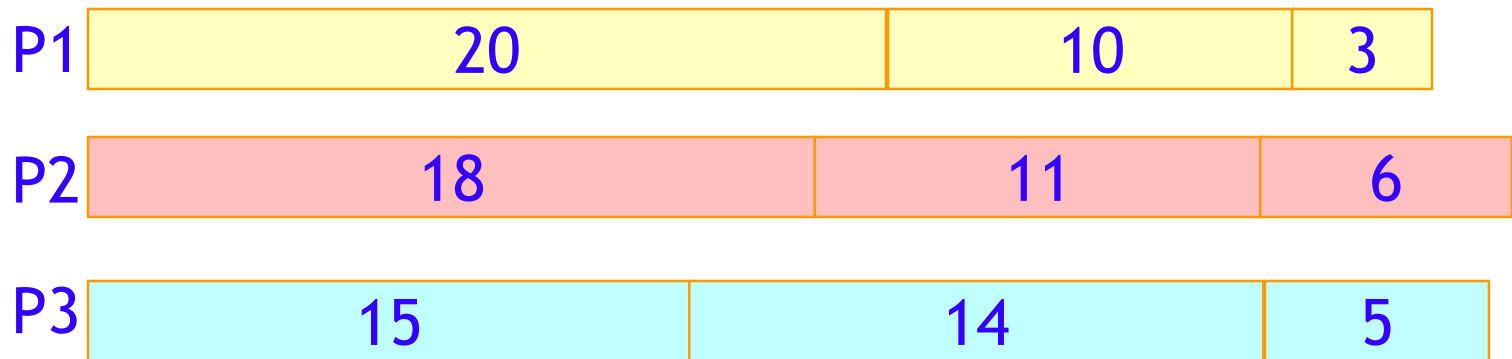
---

- In some (fictional) monetary system, “krons” come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
  - This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
  - This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution



# A scheduling problem

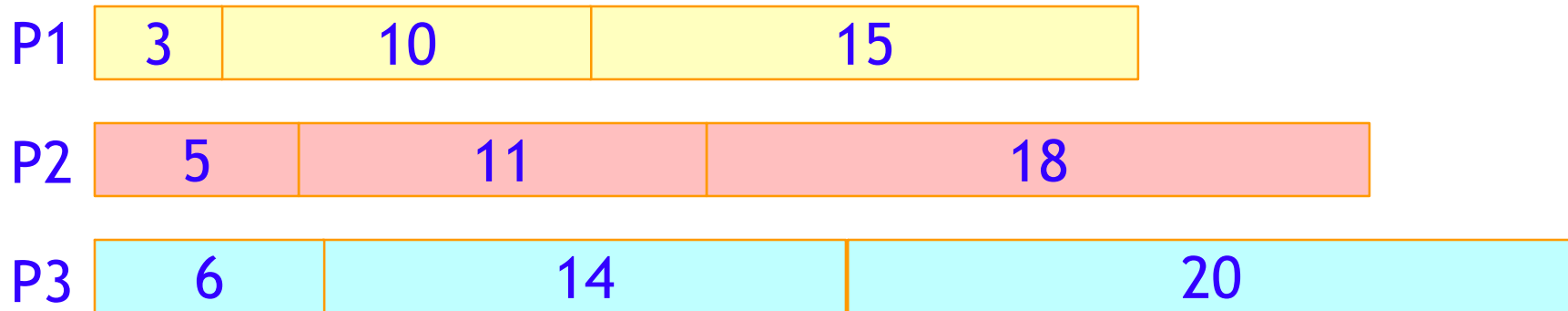
- You have to run nine jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes
- You have three processors on which you can run these jobs
- You decide to do the longest-running jobs first, on whatever processor is available



- Time to completion:  $18 + 11 + 6 = 35$  minutes
- This solution isn't bad, but we might be able to do better

# Another approach

- What would be the result if you ran the *shortest* job first?
- Again, the running times are 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes

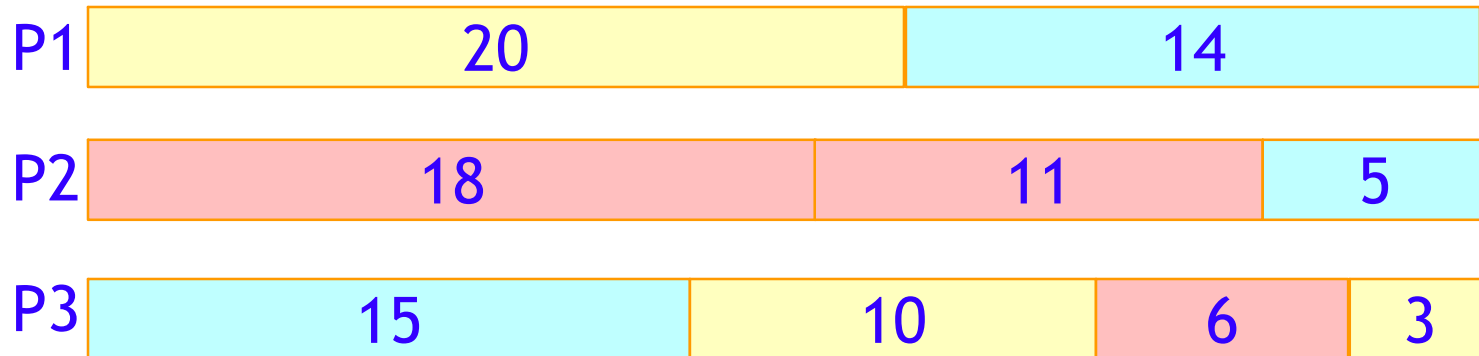


- That wasn't such a good idea; time to completion is now  $6 + 14 + 20 = 40$  minutes
- Note, however, that the greedy algorithm itself is fast
  - All we had to do at each stage was pick the minimum or maximum



# An optimum solution

- Better solutions do exist:



- This solution is clearly optimal (why?)
- Clearly, there are other optimal solutions (why?)
- How do we find such a solution?
  - One way: Try all possible assignments of jobs to processors
  - Unfortunately, this approach can take exponential time



# Huffman Coding

Prepared by Dan Stevenson  
University of Wisconsin

- Huffman codes can be used to compress information
  - Like WinZip – although WinZip doesn't use the Huffman algorithm
  - JPEGs do use Huffman as part of their compression process
- The basic idea is that instead of storing each character in a file as an 8-bit ASCII value, we will instead store the more frequently occurring characters using fewer bits and less frequently occurring characters using more bits
  - On average this should decrease the filesize (usually  $\frac{1}{2}$ )

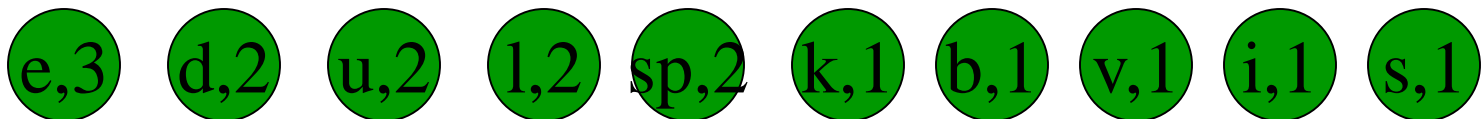




# Huffman Coding

---

- As an example, lets take the string:  
    **"duke blue devils"**
- We first do a frequency count of the characters:
  - e:3, d:2, u:2, l:2, space:2, k:1, b:1, v:1, i:1, s:1
- Next we use a Greedy algorithm to build up a Huffman Tree
  - We start with nodes for each character





# Huffman Coding

---

- We then pick the nodes with the smallest frequency and combine them together to form a new node
  - The selection of these nodes is the Greedy part
- The two selected nodes are removed from the set, but replace by the combined node
- This continues until we have only 1 node left in the set



# Huffman Coding

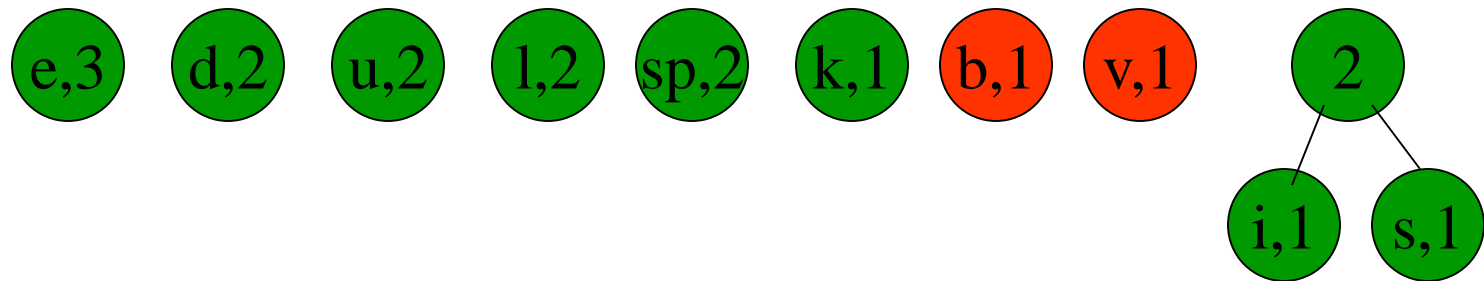
---

e,3   d,2   u,2   l,2   sp,2   k,1   b,1   v,1   i,1   s,1



# Huffman Coding

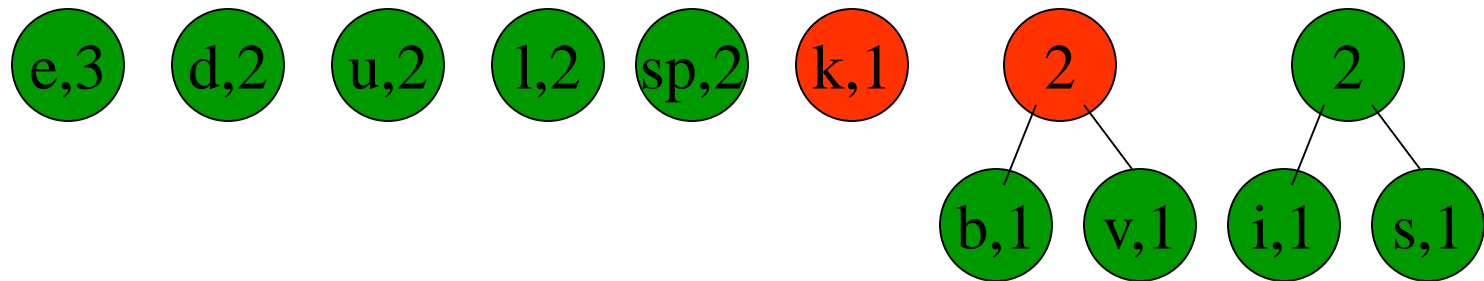
---





# Huffman Coding

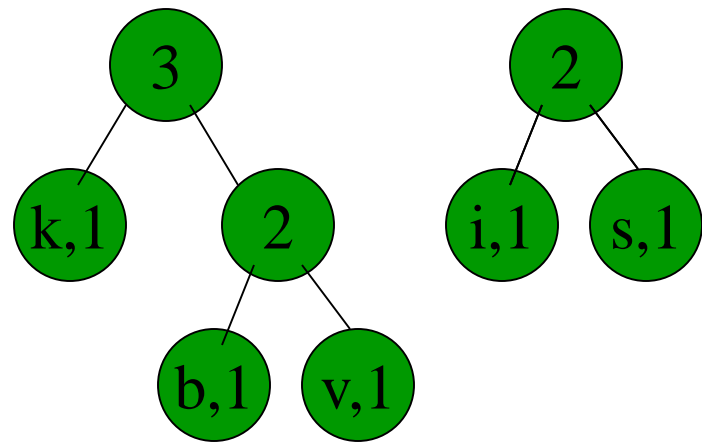
---



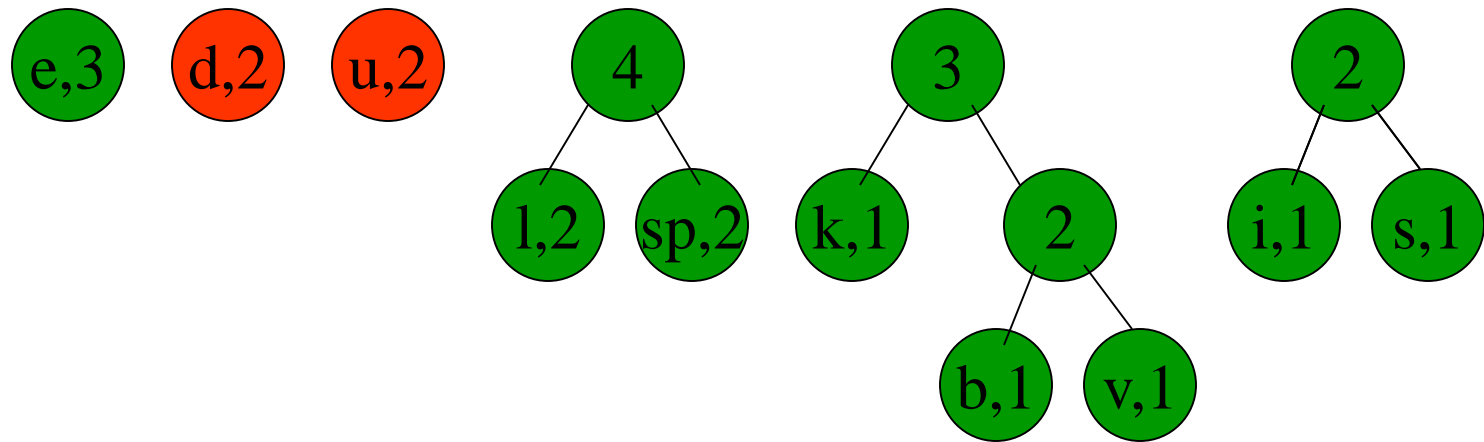


# Huffman Coding

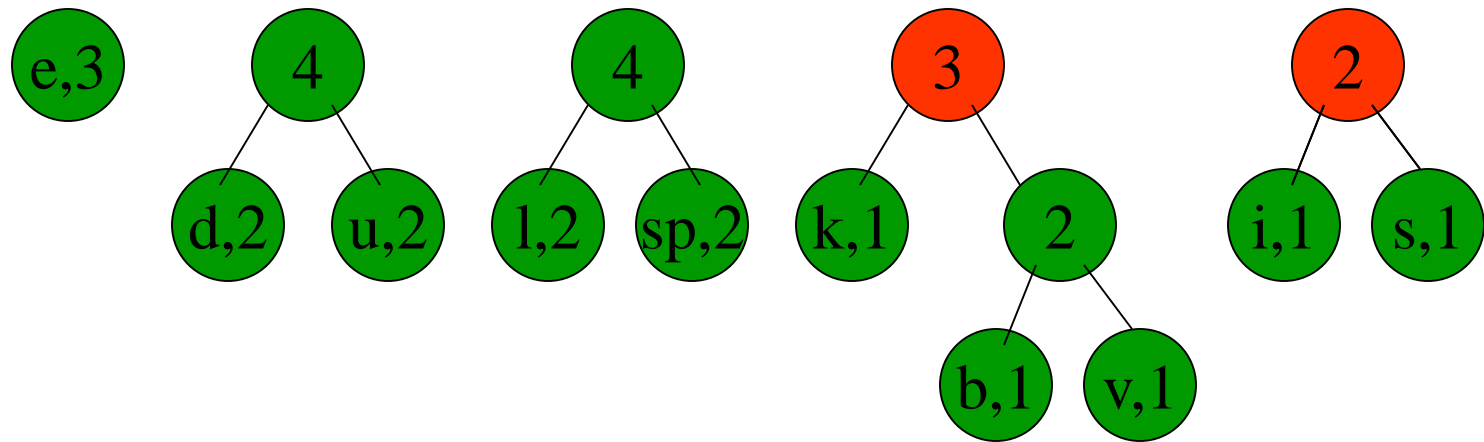
---



# Huffman Coding

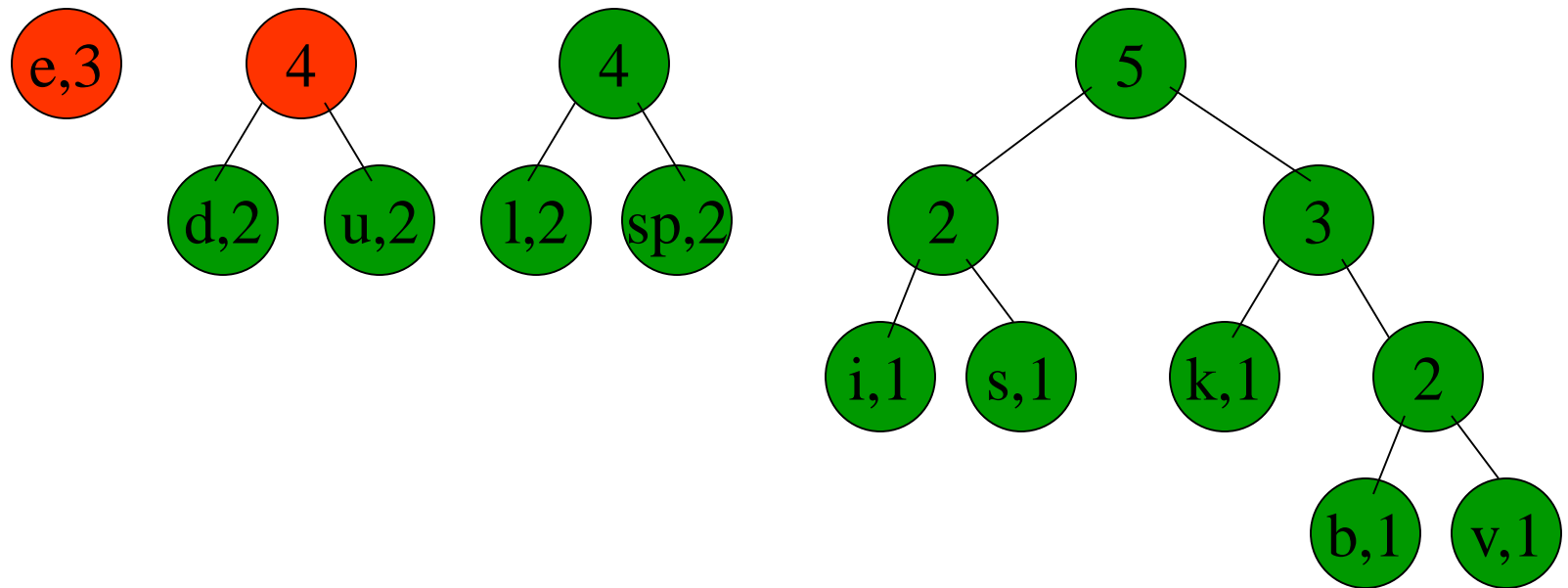


# Huffman Coding

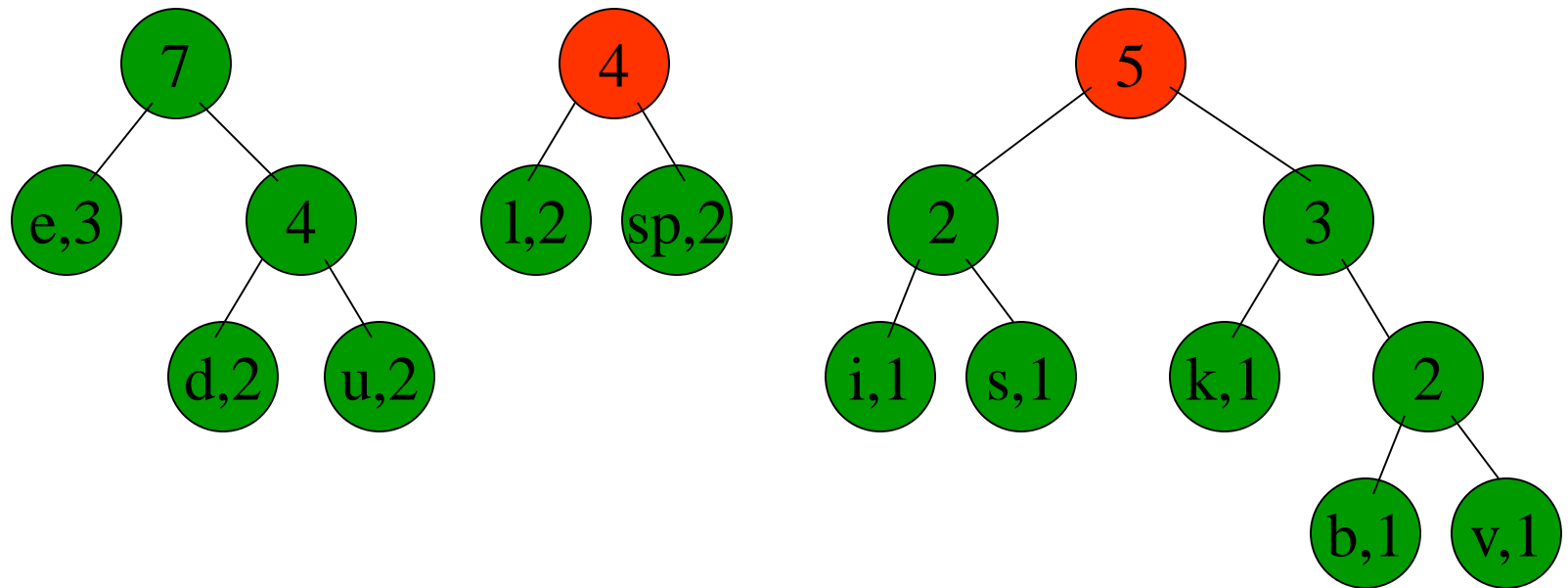




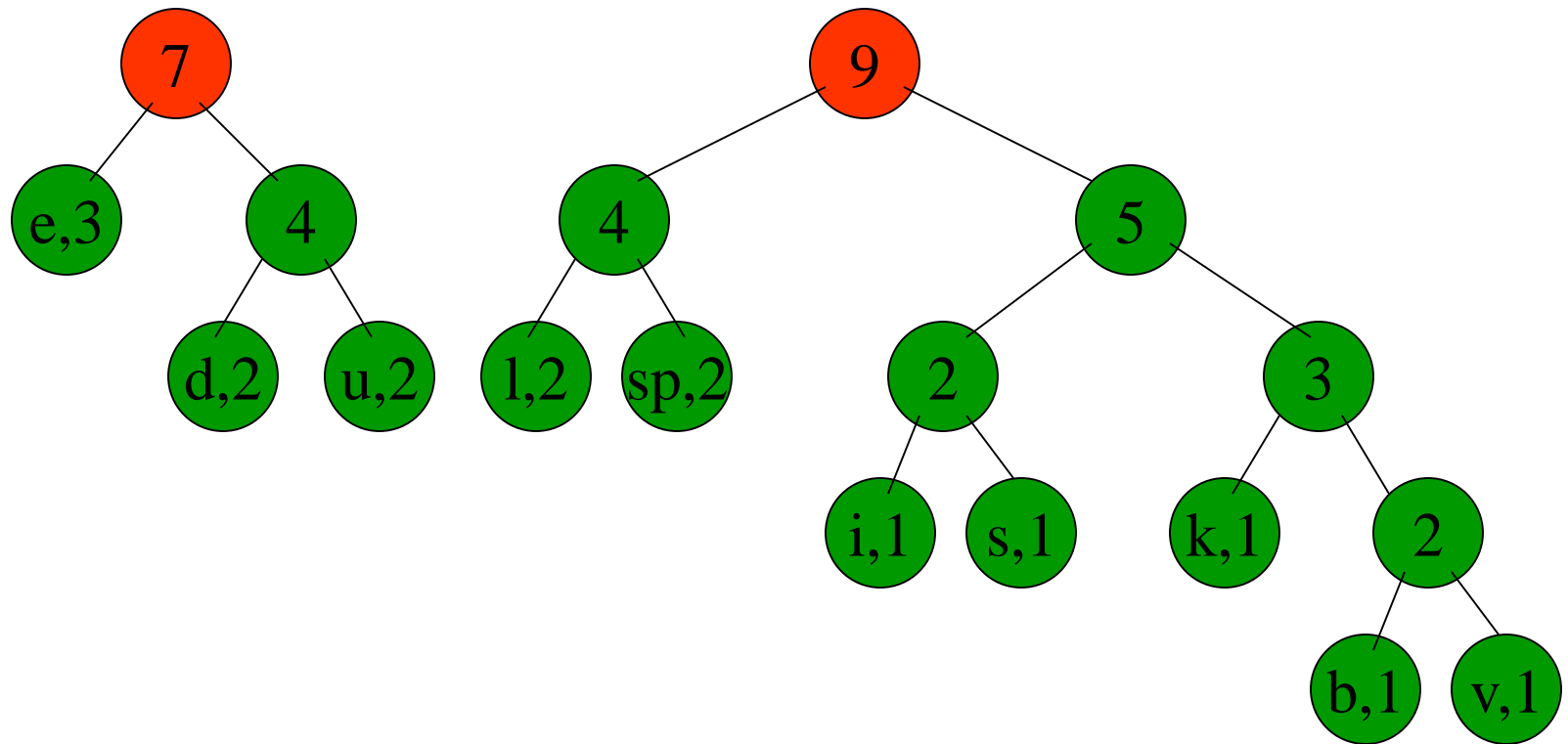
# Huffman Coding



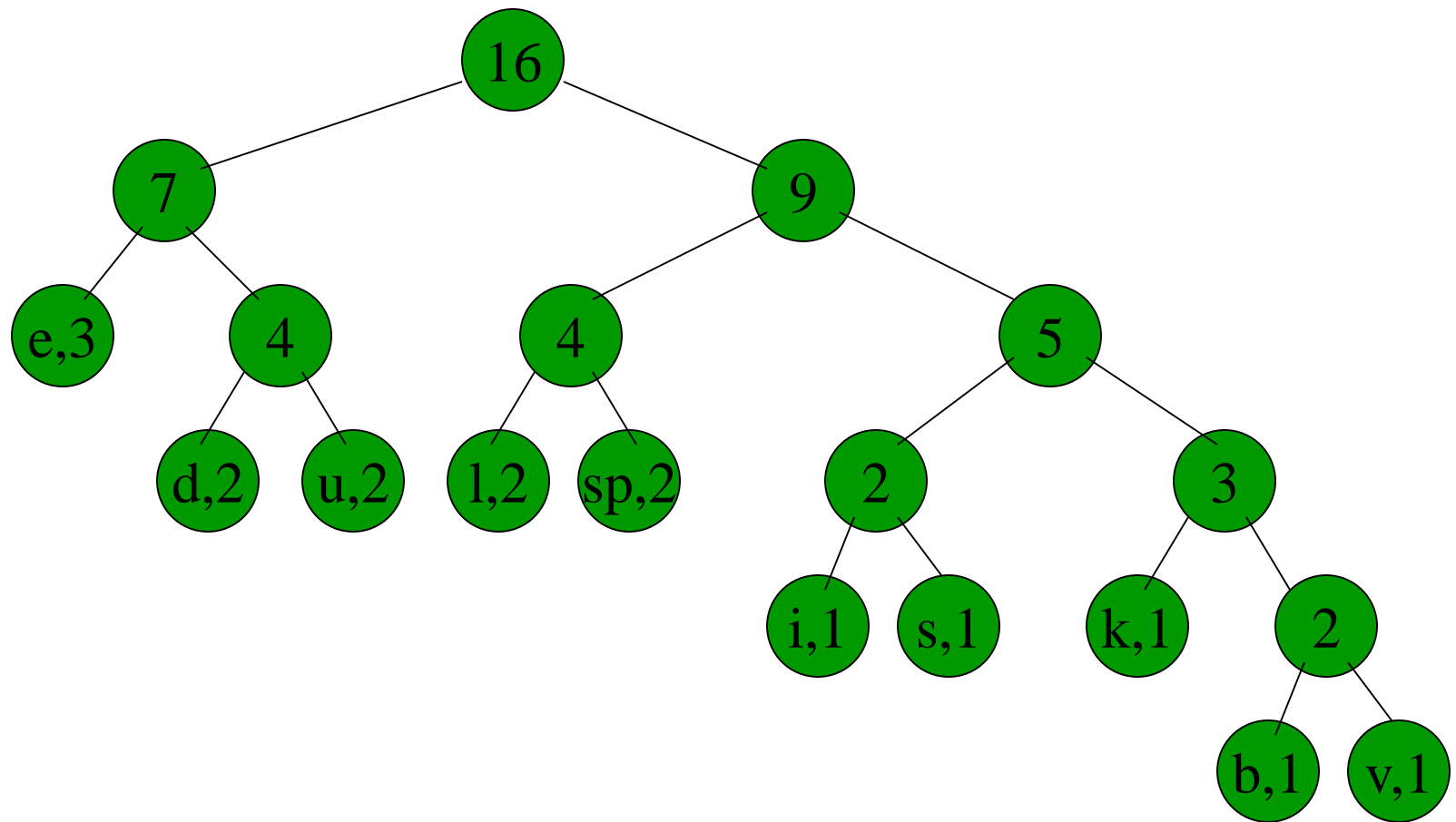
# Huffman Coding



# Huffman Coding



# Huffman Coding



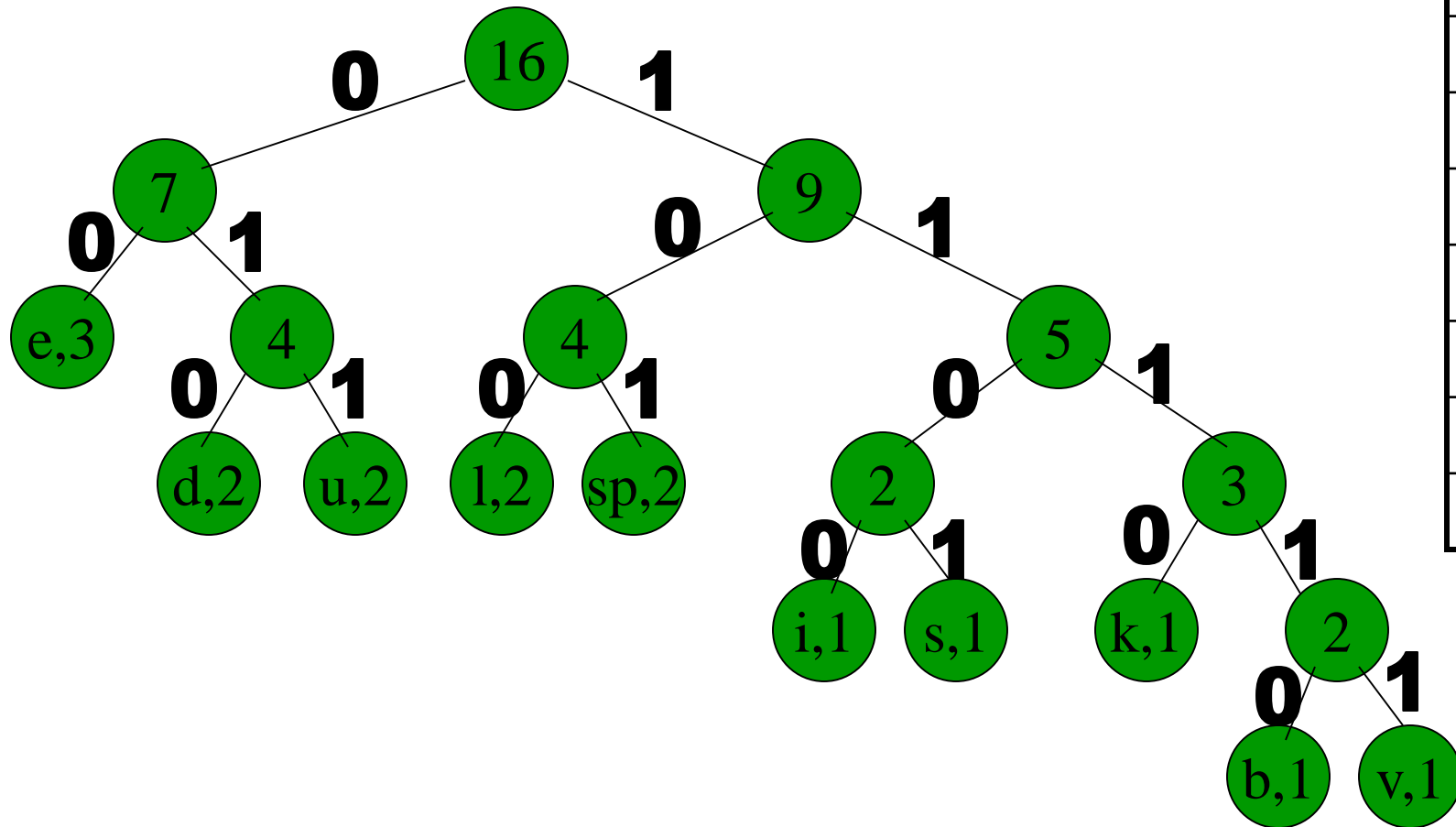


# Huffman Coding

---

- Now we assign codes to the tree by placing a 0 on every left branch and a 1 on every right branch
- A traversal of the tree from root to leaf give the Huffman code for that particular leaf character
- Note that no code is the prefix of another code

# Huffman Coding



e	00
d	010
u	011
l	100
sp	101
i	1100
s	1101
k	1110
b	11110
v	11111



# Huffman Coding

---

- These codes are then used to encode the string

- Thus, “duke blue devils” turns into:

010 011 1110 00 101 11110 100 011 00 101 010 00 11111 1100 100 1101

- When grouped into 8-bit bytes:

01001111 10001011 11101000 11001010 10001111 11100100 1101xxxx

- Thus it takes 7 bytes of space compared to 16 characters \* 1 byte/char = 16 bytes uncompressed



# Huffman Coding

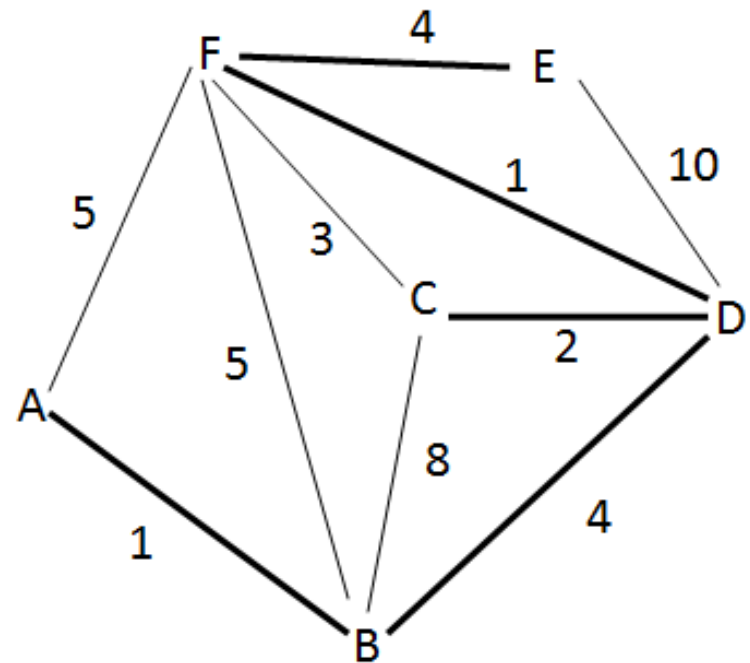
---

- Uncompressing works by reading in the file bit by bit
  - Start at the root of the tree
  - If a 0 is read, head left
  - If a 1 is read, head right
  - When a leaf is reached decode that character and start over again at the root of the tree
- Thus, we need to save Huffman table information as a header in the compressed file
  - Doesn't add a significant amount of size to the file for large files (which are the ones you want to compress anyway)
  - Or we could use a fixed universal set of codes/frequencies



# Minimum Spanning Tree

- Input: graph  $G$  with weights on the edges
- Output: connected sub graph  $G'$  of  $G$  that includes all the vertices of  $G$ , of minimum total weight





# Exhaustive Search

---

- List all connected sub-graphs of  $G$
- Return sub-graph with least weight
- Number of vertices,  $N$ , and number of edges,  $M$
- $O(m^{n-1})$



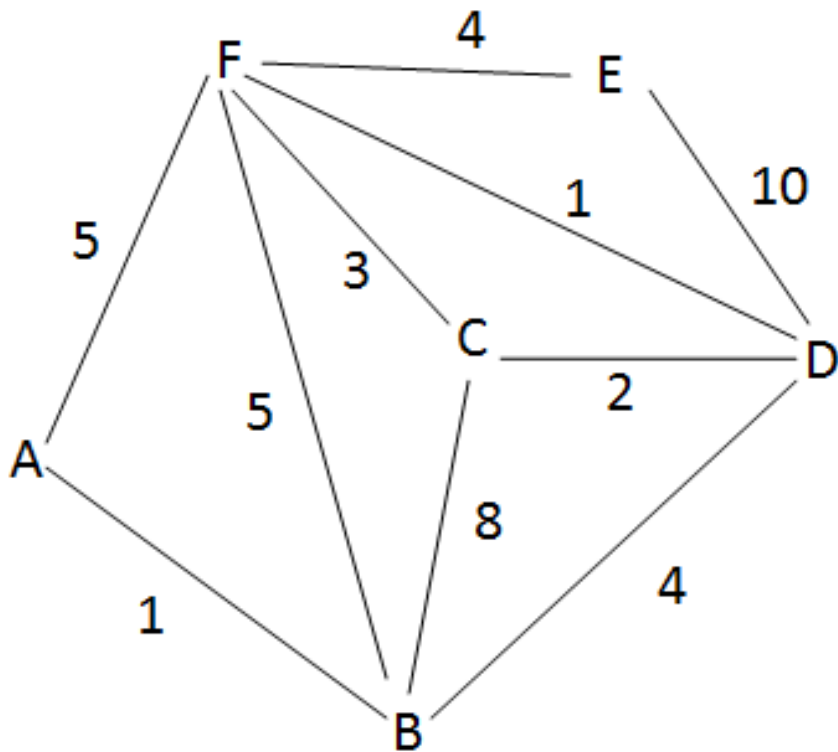
# Greedy Algorithm

---

- Start with a graph containing just the vertices,  
 $G' = \{V, \emptyset\}$
- Add edges with the least weight until the graph is connected but no cycles are created
- To do this:
  - Order the edges in increasing weight
  - While the graph is not connected, add an edge
  - End when all vertices are connected

# Greedy Algorithm - Example

**Initial Graph:**



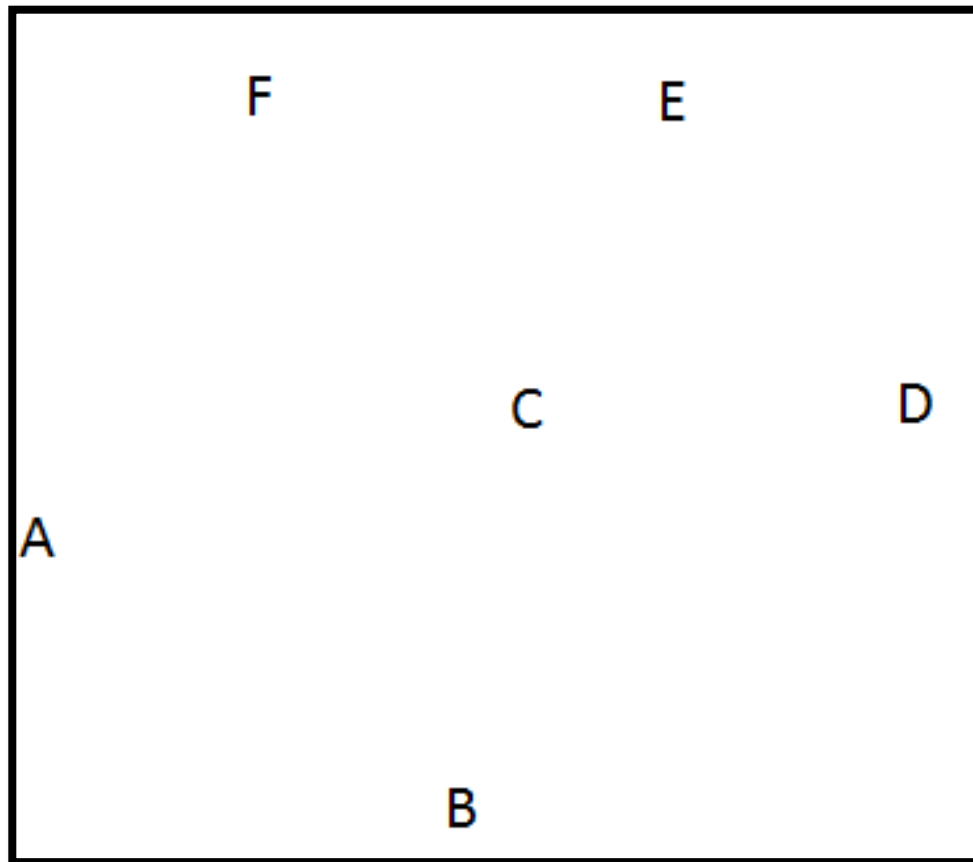
**List of Edges:**

Edge	Weight
AB	1
AF	5
BC	8
BD	4
BF	5
CD	2
CF	3
DE	10
DF	1
EF	4

# Greedy Algorithm - Example

Start with  $G' = \{V, \emptyset\}$  and sorted list of edges

**Graph:**



**Sorted List of Edges:**

Edge	Weight
AB	1
DF	1
CD	2
CF	3
BD	4
EF	4
AF	5
BF	5
BC	8
DE	10

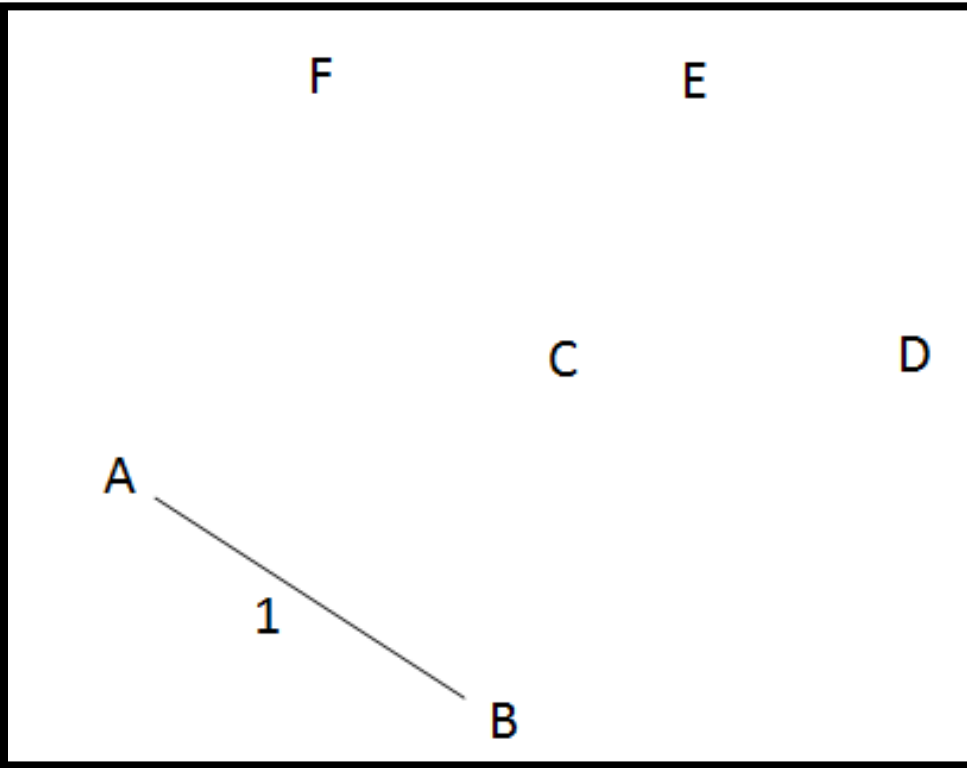
# Greedy Algorithm - Example

Add edge from list to graph s.t. no cycles are created  
Remove edge from list

**Graph:**

**Sorted List of Edges:**

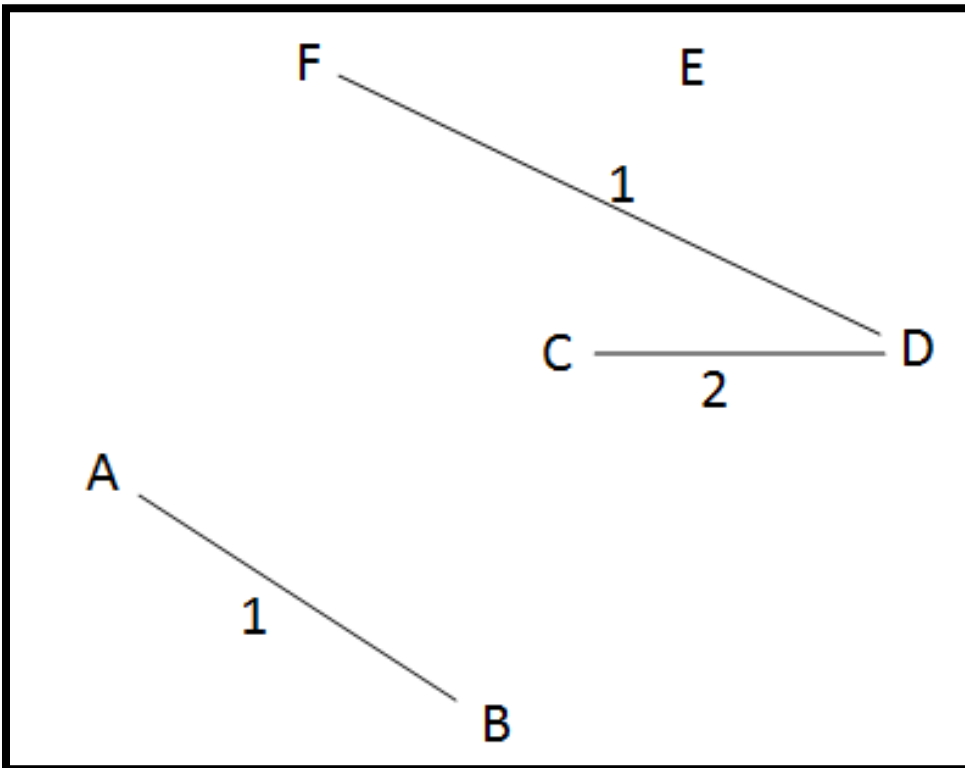
Edge	Weight
AB	1
DF	1
CD	2
CF	3
BD	4
EF	4
AF	5
BF	5
BC	8
DE	10



# Greedy Algorithm - Example

Add edge from list to graph s.t. no cycles are created  
Remove edge from list

**Graph:**



**Sorted List of Edges:**

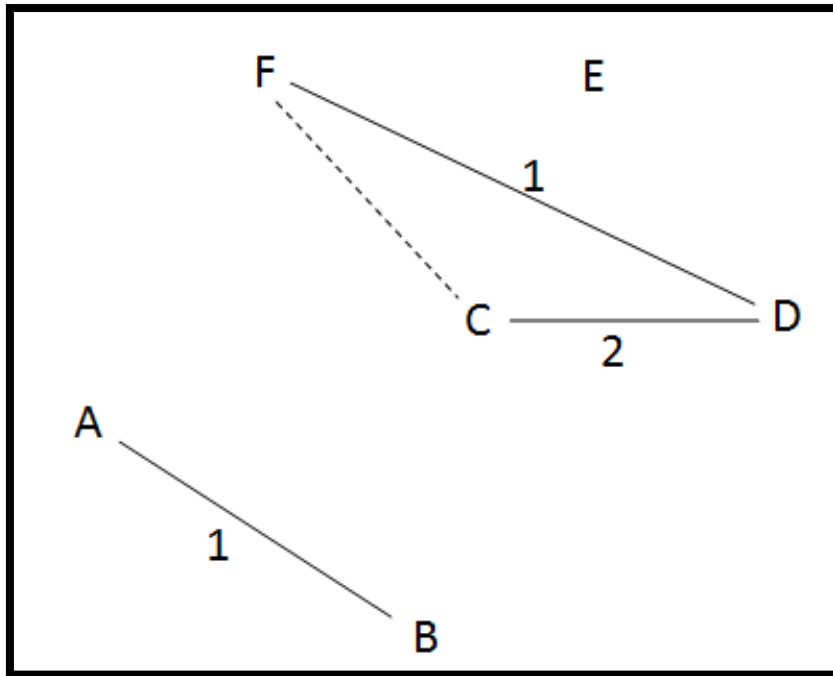
Edge	Weight
AB	1
DF	1
CD	2
CF	3
BD	4
EF	4
AF	5
BF	5
BC	8
DE	10

# Greedy Algorithm - Example

Add edge from list to graph s.t. no cycles are created

Remove edge from list

**Graph:**



**Sorted List of Edges:**

Edge	Weight
CF	3
BD	4
EF	4
AF	5
BF	5
BC	8
DE	10

Check for cycles using Depth First Search starting at a vertex in the added edge:

- Start at C
- C → D → F → C
- C gets visited twice ⇒ a cycle exists and CF should not be added to the graph



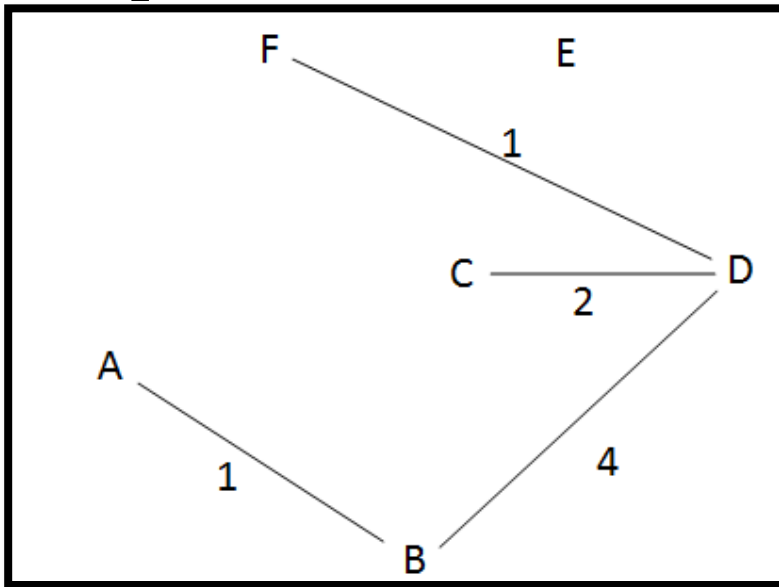
# Greedy Algorithm - Example

Add edge from list to graph s.t. no cycles are created

Remove edge from list

End when all vertices can be visited (graph is connected)

**Graph:**



**Sorted List of Edges:**

Edge	Weight
BD	4
EF	4
AF	5
BF	5
BC	8
DE	10

Check for cycles using Depth First Search starting at a vertex in the added edge:

- Start at B
- B → D → C → F → A
- If all vertices are visited by DFS, then the graph is connected and we are done

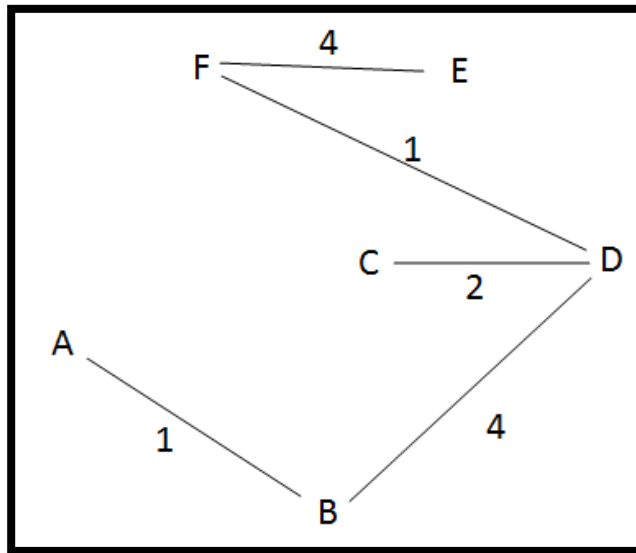
# Greedy Algorithm - Example

Add edge from list to graph s.t. no cycles are created

Remove edge from list

End when all vertices can be visited (graph is connected)

**Graph:**



**Sorted List of Edges:**

Edge	Weight
EF	4
AF	5
BF	5
BC	8
DE	10

Check for cycles using Depth First Search starting at a vertex in the added edge:

- Start at E
- E -> F -> D -> B -> A -> C
- All vertices were visited and there were no cycles => we have found a minimum spanning tree with weight 12.



# Greedy Algorithm- Pseudocode

---

- Given  $G = (V, E)$
- $G' \leftarrow (V, \emptyset)$
- While  $G'$  is not connected
  - Add  $e \in E$  to  $G'$  s.t.  $G'$  is acyclic and  $e$  is minimum
  - Remove  $e$  from  $E$



# Greedy Algorithm – Run Time

---

- Initialization – constant
- While loop –  $O(n-1)$ 
  - Connected graph has  $n-1$  edges
  - Must add  $n-1$  edges to the graph for it to be connected
- Find a minimum  $e \in E$  –  $O(m)$
- Make sure  $G'$  is acyclic –  $O(2n)$ 
  - DFS is  $O(m+n) < O(2n)$  where  $m \leq n-1$
- Test connectivity of  $G'$  –  $O(n)$ 
  - Can use DFS; could be done in same step as testing acyclicity
- Remove  $e$  from  $E$  - constant
- Total Runtime:  $O(n \cdot (m + 2n + n)) \sim O(nm + n^2) \rightarrow$   
POLYNOMIAL

Horrible Bad Fair Good Excellent

Operations

$O(n!)$

$O(2^n)$

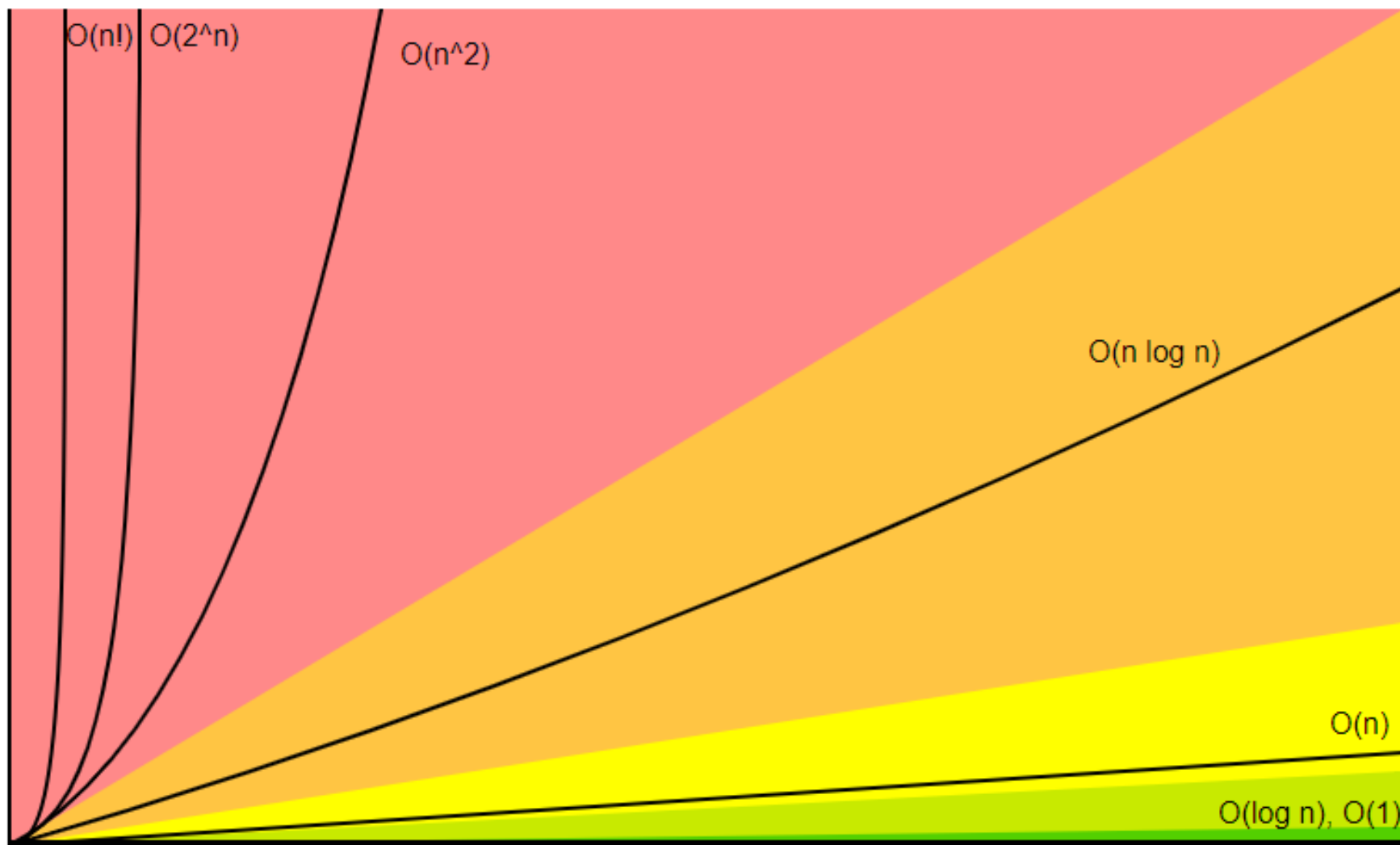
$O(n^2)$

$O(n \log n)$

$O(n)$

$O(\log n), O(1)$

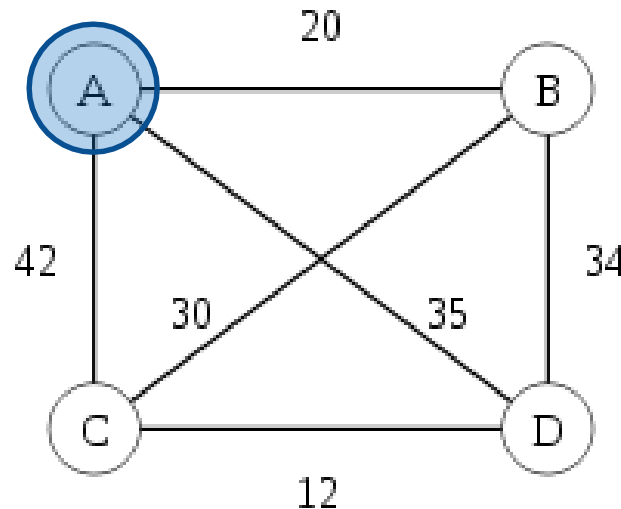
Elements



# Traveling Salesman Problem (TSP)

Prepared by Caisar Oentoro

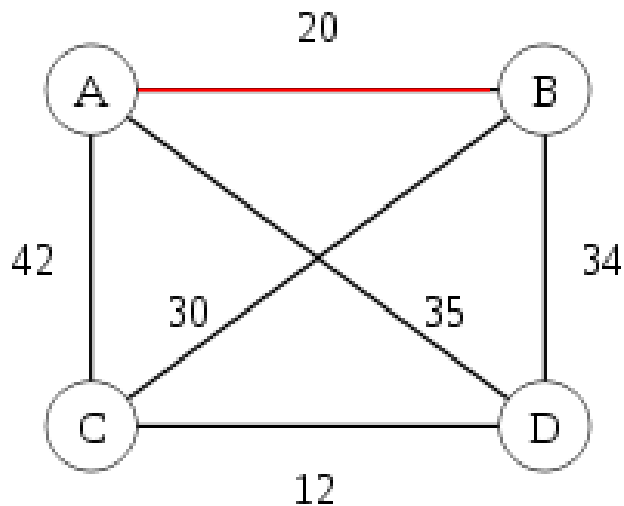
- > The Problem is how to travel from city A and visit all city on the map, then back to city A again.



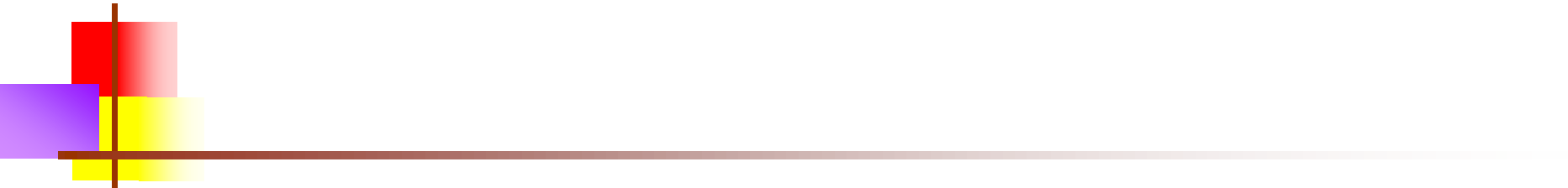
- > The rules: you only visit each city once and you can't pass through any traversed path.

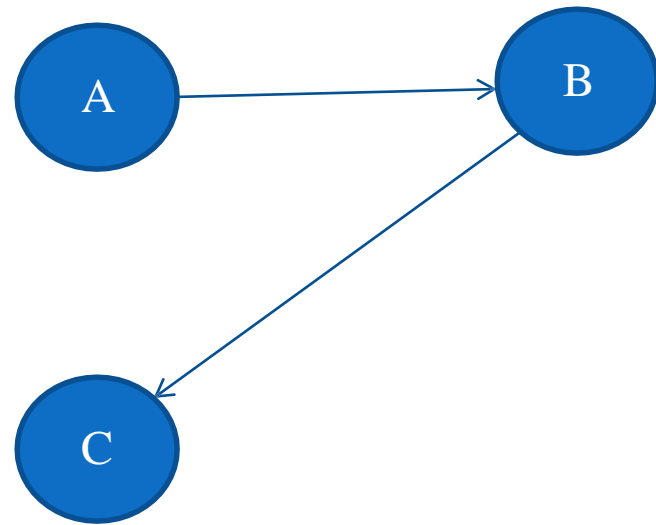
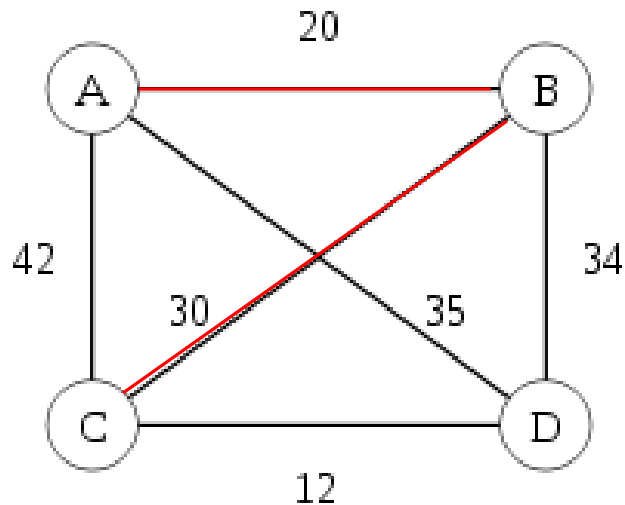
# Solution:

- > Find the shortest path from city A(start) to any other city.

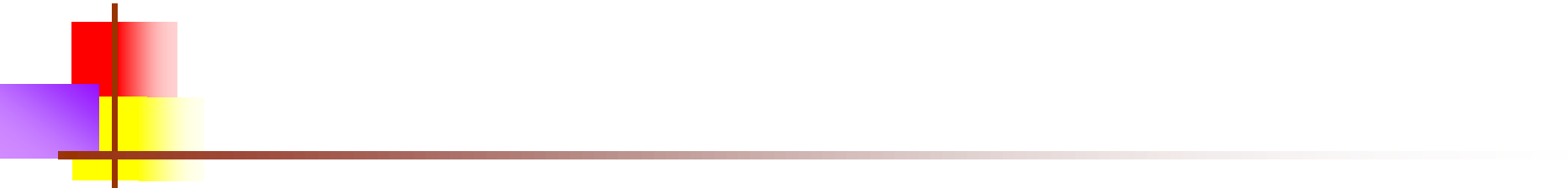


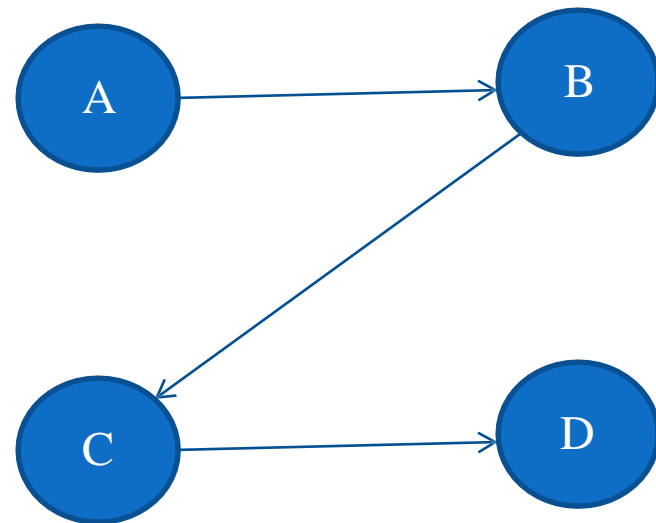
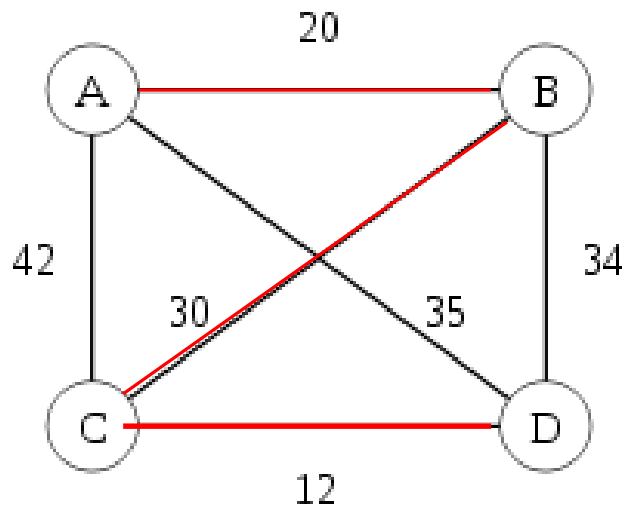
- > Because the nearest city is B, so we go to B

- 
- > From B, we find any other city but A (because A has been visited) that has nearest path. So we choose C:

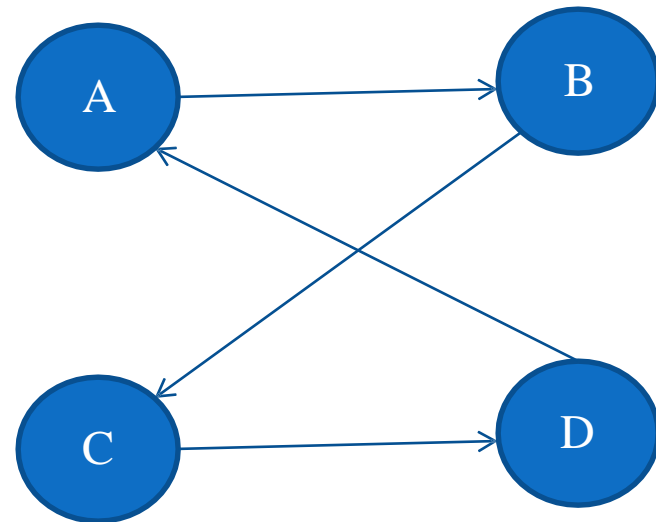
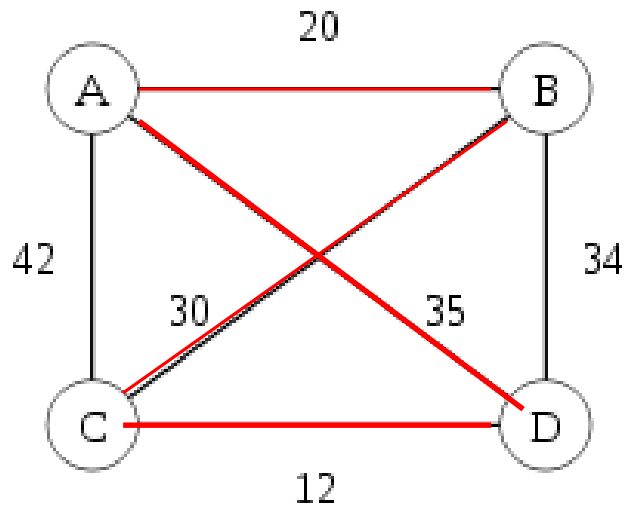




- 
- > From C, we look to nearest city again, but don't look for A and B, because both has been visited. So we choose D.



> At this node(D), we can't go to any city, because all neighbor of D has been visited. We go back to first city(A).



> And that was how to solve TSP problem.

# Advantage of Greedy

- > Greedy is easy to be implemented. Just search the best choice from the current state that 'reachable' (has any paths or any connections).
- > In simple case, greedy often give you the best solution.





# Drawback of Greedy

---

- > In large and complex case, greedy doesn't always give you the best solution, because it's just search and take the best choice that you can reach from the current state.
- > It takes longer time than any other algorithms for big case of problem



# Analysis

---

- A greedy algorithm typically makes (approximately)  $n$  choices for a problem of size  $n$ 
  - (The first or last choice may be forced)
- Hence the expected running time is:  
 $O(n * O(\text{choice}(n)))$ , where  $\text{choice}(n)$  is making a choice among  $n$  objects
  - Counting: Must find largest useable coin from among  $k$  sizes of coin ( $k$  is a constant), an  $O(k)=O(1)$  operation;
    - Therefore, coin counting is  $(n)$
  - Huffman: Must sort  $n$  values before making  $n$  choices
    - Therefore, Huffman is  $O(n \log n) + O(n) = O(n \log n)$
  - Minimum spanning tree: At each new node, must include new edges and keep them sorted, which is  $O(n \log n)$  overall
    - Therefore, MST is  $O(n \log n) + O(n) = O(n \log n)$



# Other greedy algorithms

---

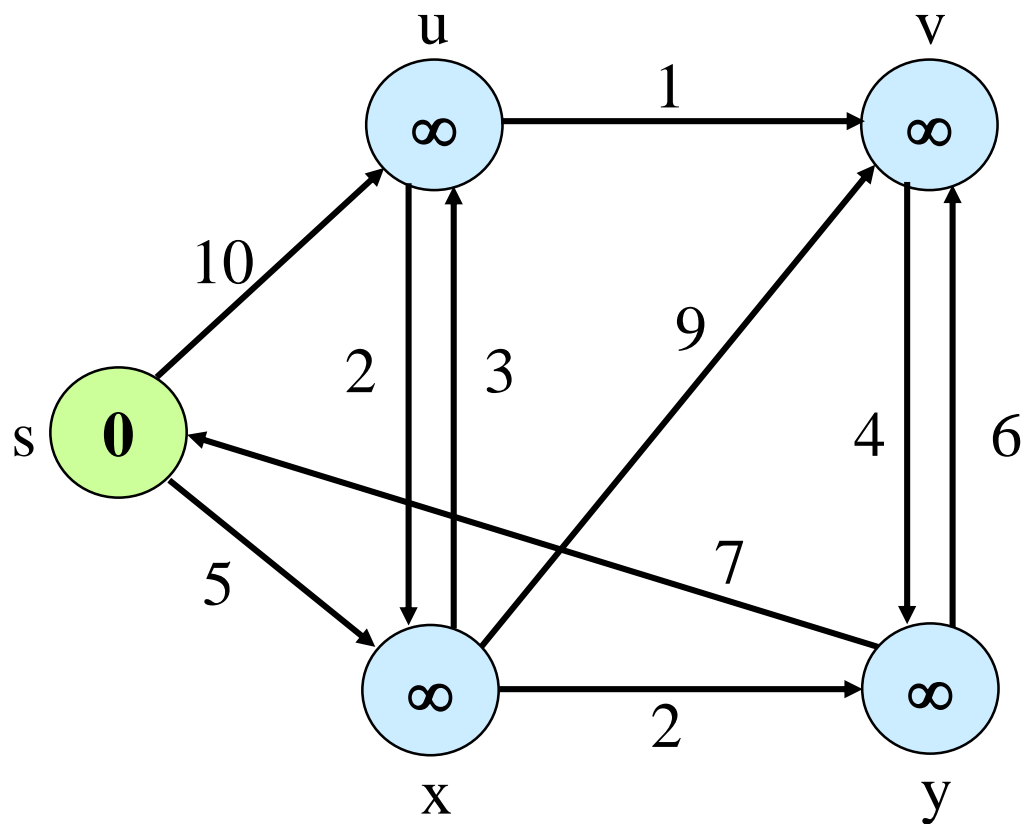
- Dijkstra's algorithm for finding the shortest path in a graph
  - Always takes the *shortest* edge connecting a known node to an unknown node
- Kruskal's algorithm for finding a minimum-cost spanning tree
  - Always tries the *lowest-cost* remaining edge
- Prim's algorithm for finding a minimum-cost spanning tree
  - Always takes the *lowest-cost* edge between nodes in the spanning tree and nodes not yet in the spanning tree



# Dijkstra's shortest-path algorithm

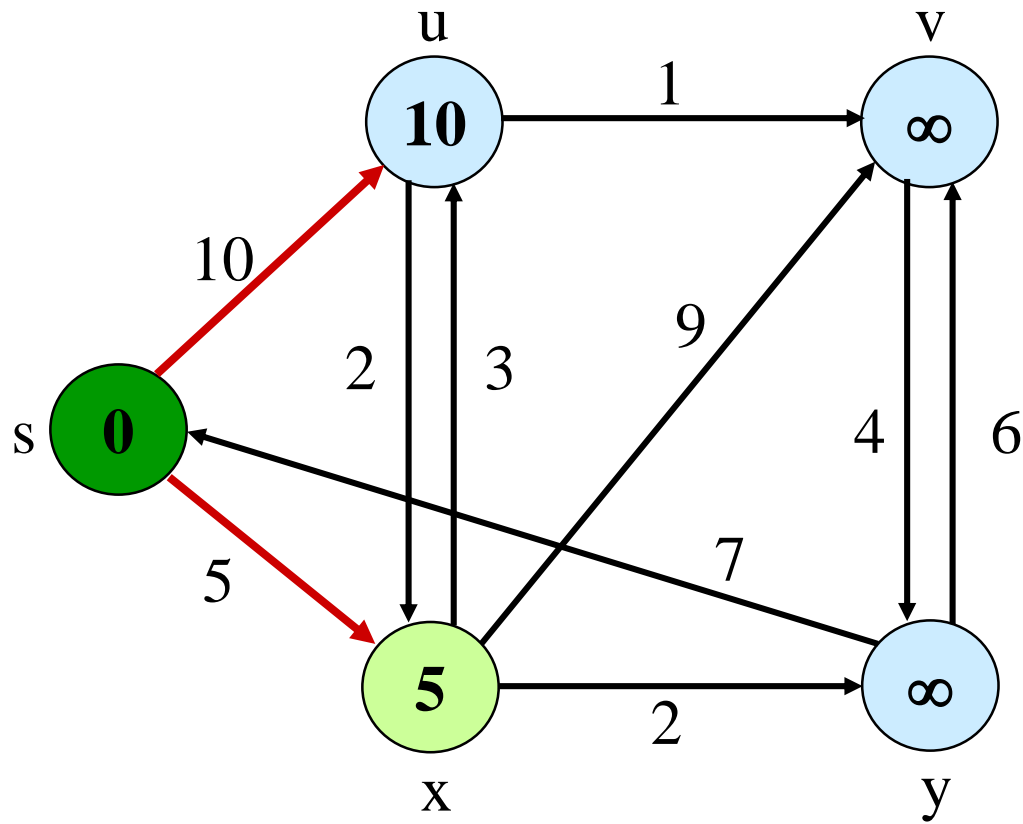
- Dijkstra's algorithm finds the shortest paths from a given node to all other nodes in a graph
  - Initially,
    - Mark the given node as *known* (path length is zero)
    - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
  - Repeatedly (until all nodes are known),
    - Find an unknown node containing the smallest distance
    - Mark the new node as known
    - For each node adjacent to the new node, examine its neighbors to see whether their estimated distance can be reduced (distance to known node plus cost of out-edge)
      - If so, also reset the predecessor of the new node

# Example

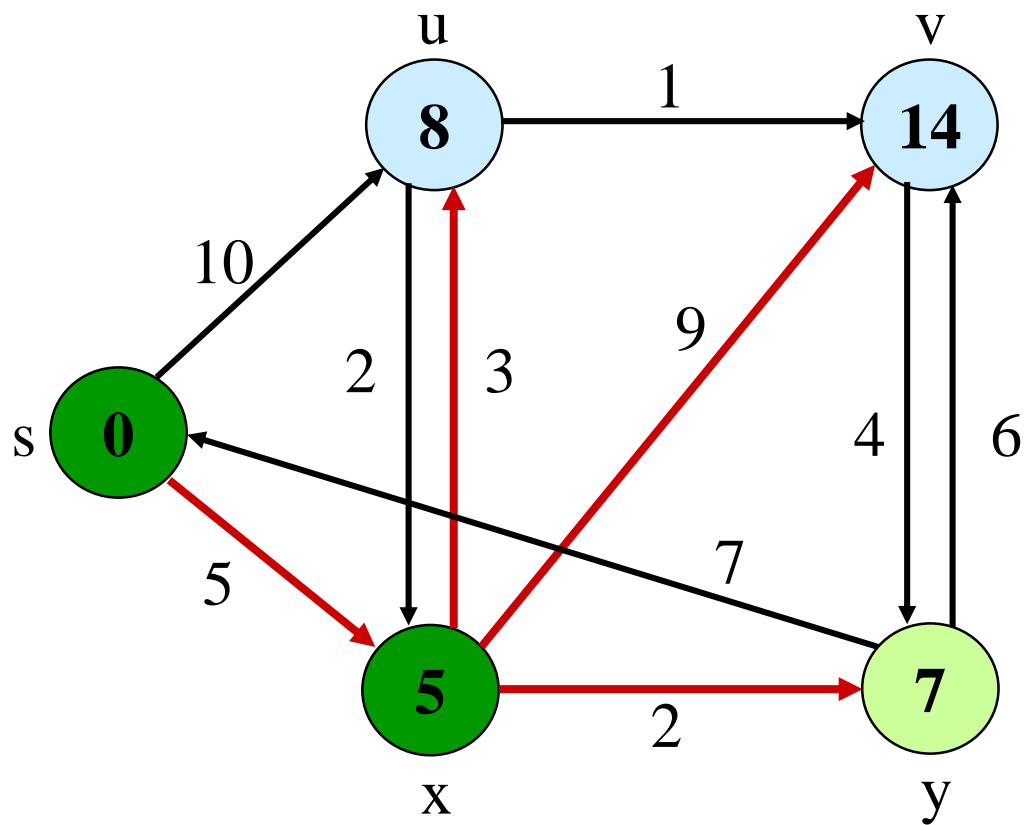




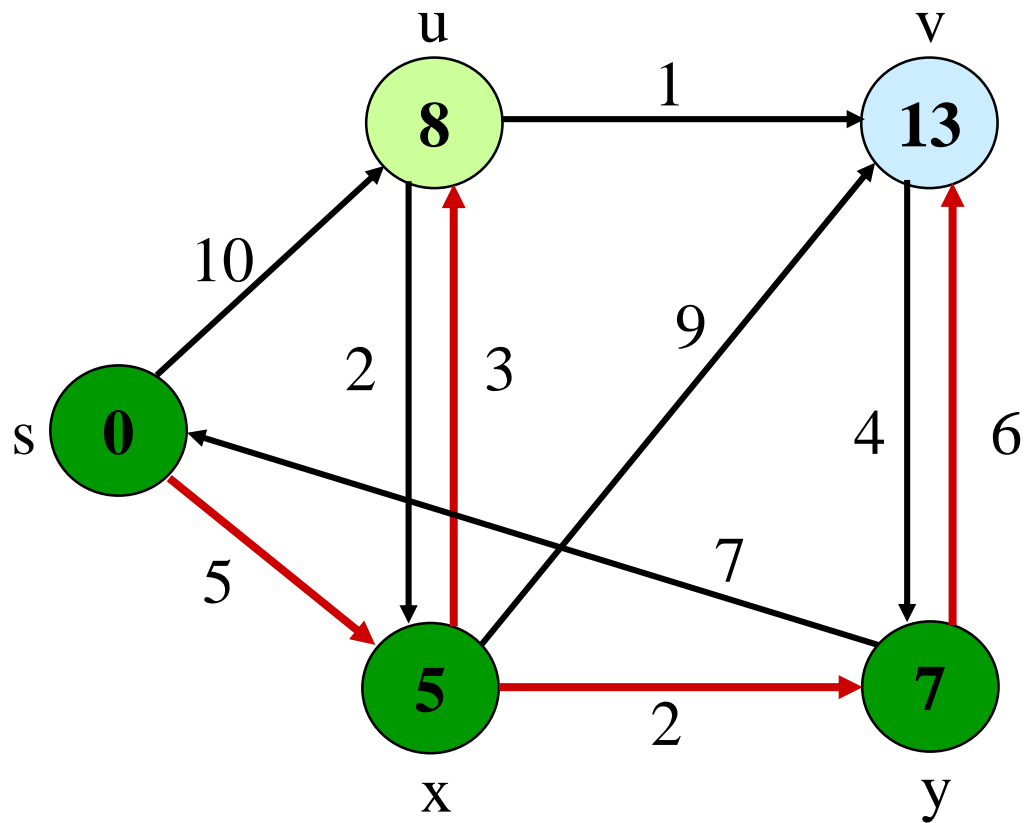
# Example



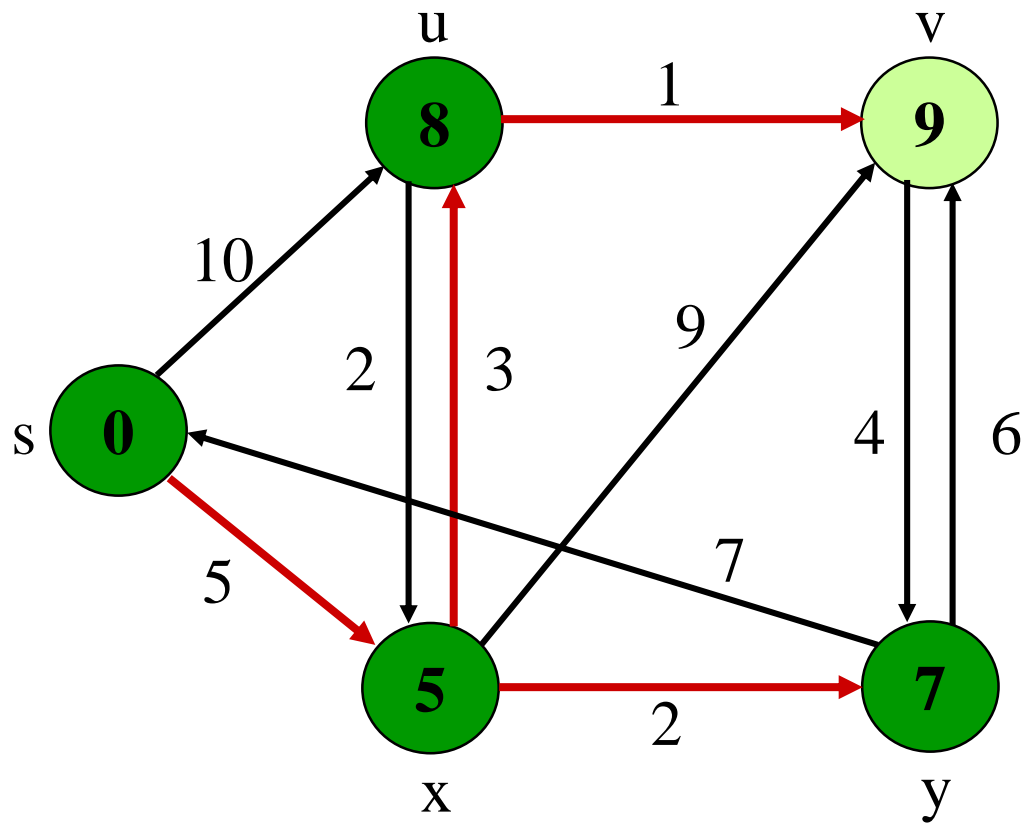
# Example



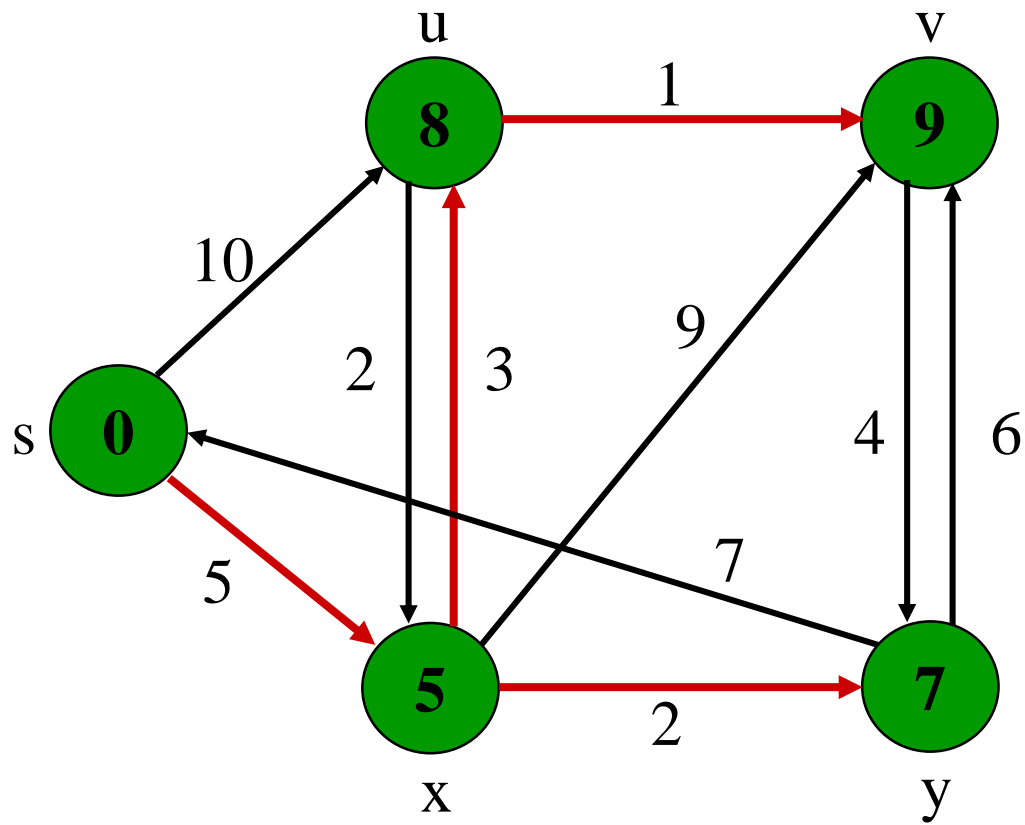
# Example



# Example



# Example





# Analysis of Dijkstra's algorithm I

- Assume that the *average* out-degree of a node is some constant  $k$ 
  - Initially,
    - Mark the given node as *known* (path length is zero)
      - This takes  $O(1)$  (constant) time
    - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node
      - If each node refers to a list of  $k$  adjacent node/edge pairs, this takes  $O(k) = O(1)$  time, that is, constant time
      - Notice that this operation takes *longer* if we have to extract a list of names from a hash table



# Analysis of Dijkstra's algorithm II

- Repeatedly (until all nodes are known), ( $n$  times)
  - Find an unknown node containing the smallest distance
    - Probably the best way to do this is to put the unknown nodes into a priority queue; this takes  $k * O(\log n)$  time *each* time a new node is marked “known” (and this happens  $n$  times)
  - Mark the new node as known --  $O(1)$  time
  - For each node adjacent to the new node, examine its neighbors to see whether their estimated distance can be reduced (distance to known node plus cost of out-edge)
    - If so, also reset the predecessor of the new node
    - There are  $k$  adjacent nodes (on average), operation requires constant time at each, therefore  $O(k)$  (constant) time
  - Combining all the parts, we get:  
 $O(1) + n * (k * O(\log n) + O(k))$ , that is,  $O(nk \log n)$  time



# Kruskal's MST Algorithm

---

Prepared by  
Kenrick Mock

- Idea: Greedily construct the MST
  - Go through the list of edges and make a forest that is a MST
  - At each vertex, sort the edges
  - Edges with smallest weights examined and possibly added to MST before edges with higher weights
  - Edges added must be “safe edges” that do not ruin the tree property.



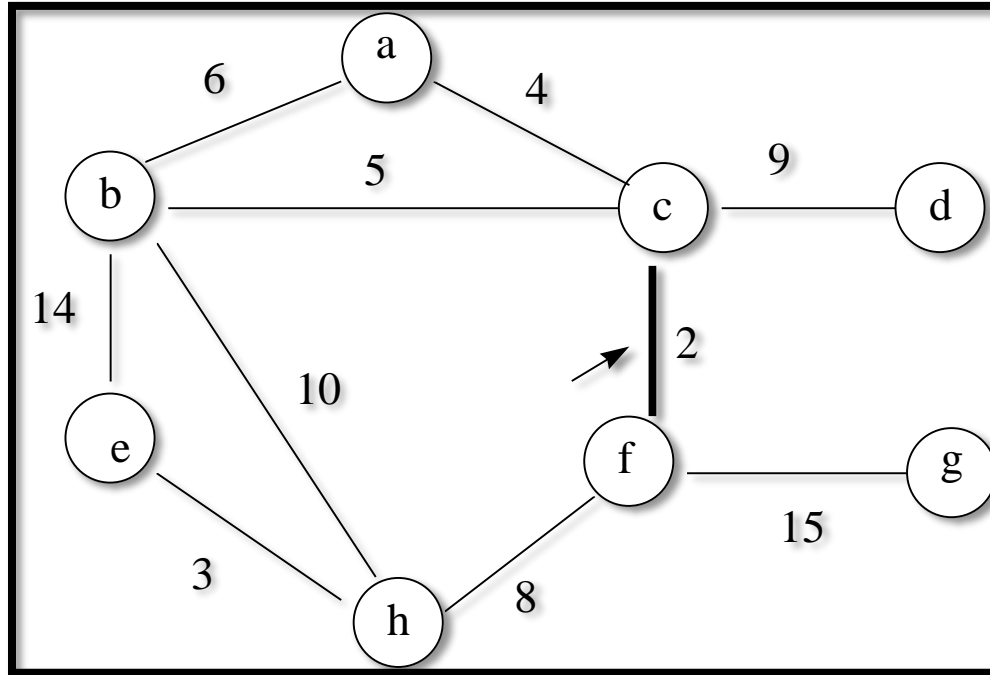


# Kruskal's Algorithm

---

```
Kruskal( $G, w$ )           ; Graph  $G$ , with weights  $w$ 
     $A \leftarrow \{\}$        ; Our MST starts empty
    for each vertex  $v \in V[G]$  do Make-Set( $v$ )    ; Make each vertex a set
    Sort edges of  $E$  by increasing weight
    for each edge  $(u, v) \in E$  in order
        ; Find-Set returns a representative (first vertex) in the set
        do if Find-Set( $u$ )  $\neq$  Find-Set( $v$ )
            then  $A \leftarrow A \cup \{(u, v)\}$ 
            Union( $u, v$ )           ; Combines two trees
    return  $A$ 
```

# Kruskal's Example



- $A = \{ \}$ , Make each element its own set.  $\{a\} \{b\} \{c\} \{d\} \{e\} \{f\} \{g\} \{h\}$
- Sort edges.
- Look at smallest edge first:  $\{c\}$  and  $\{f\}$  not in same set, add it to  $A$ , union together.
- Now get  $\{a\} \{b\} \{c f\} \{d\} \{e\} \{g\} \{h\}$

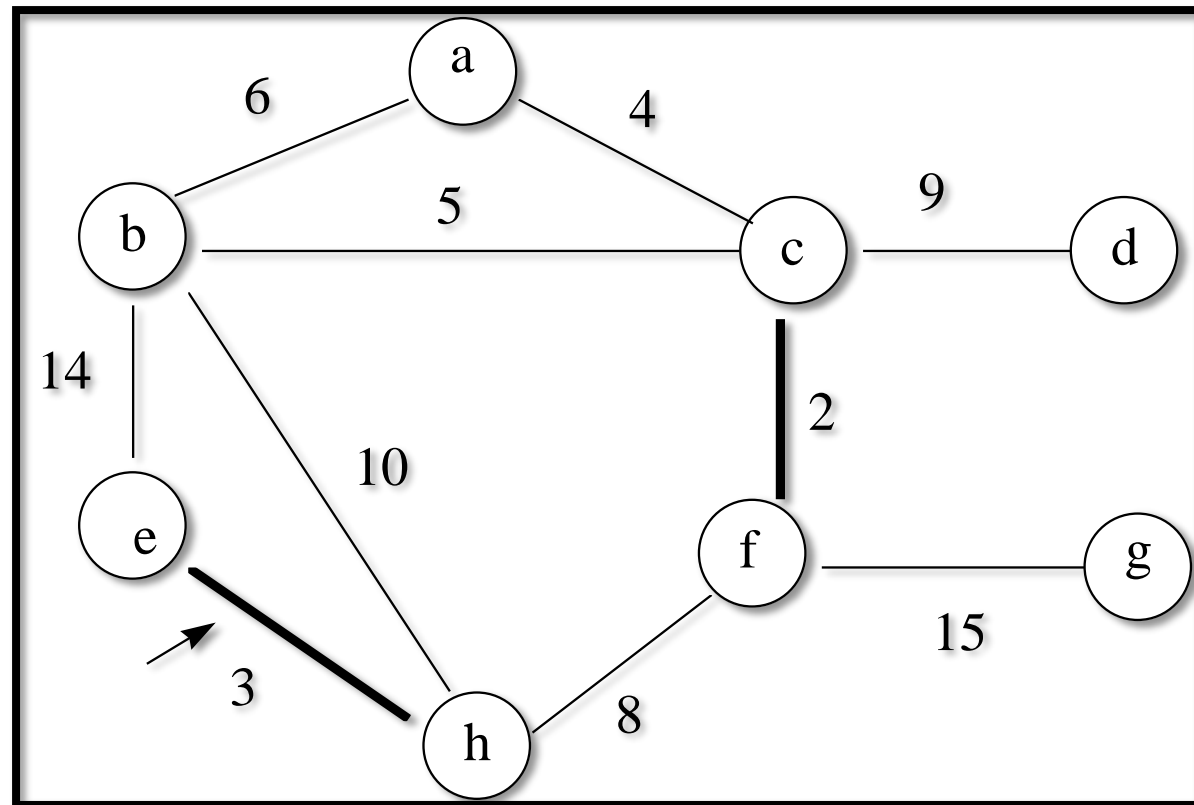
# Kruskal's Example

Keep going, checking next smallest edge.

Had: {a} {b} {c f} {d} {e} {g} {h}

{e}  $\neq$  {h}, add edge.

Now get {a} {b} {c f} {d}  
{e h} {g}



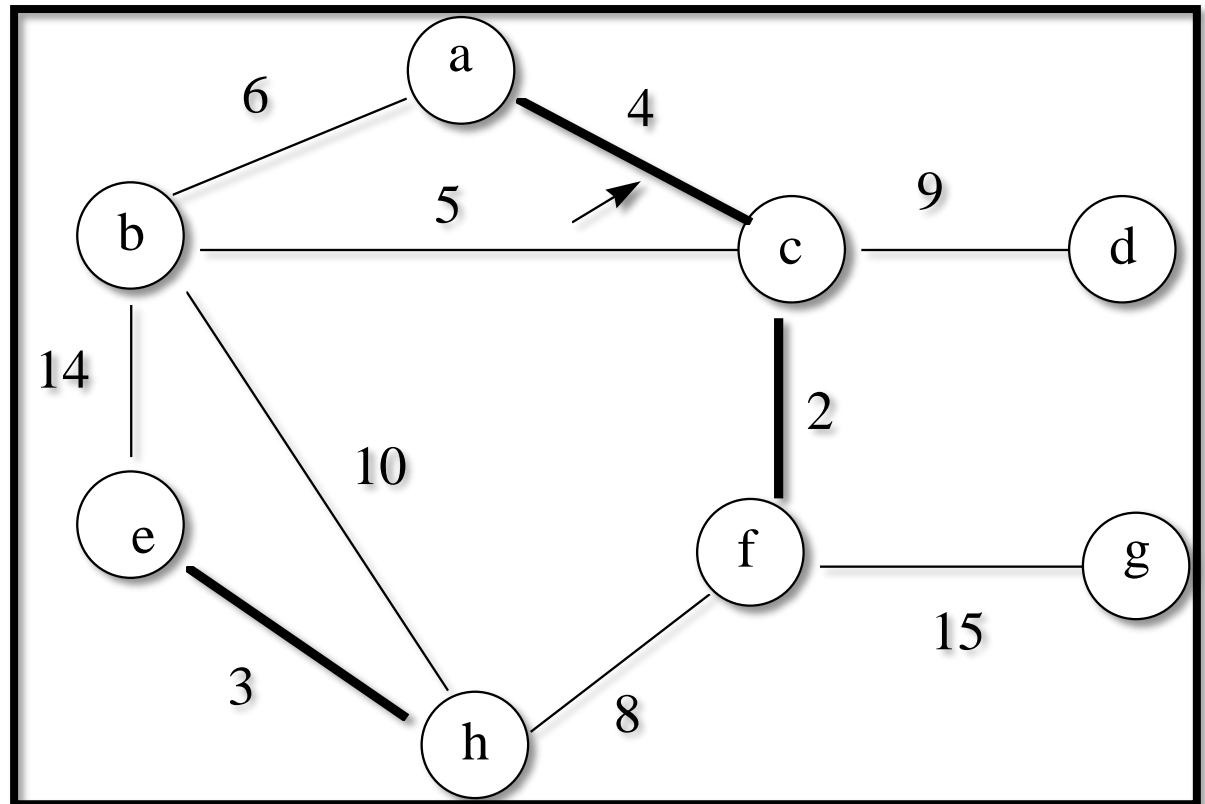
# Kruskal's Example

Keep going, checking next smallest edge.

Had: {a} {b} {c f} {d} {e h} {g}

{a}  $\neq$  {c f}, add edge.

Now get {b} {a c f}  
{d} {e h} {g}

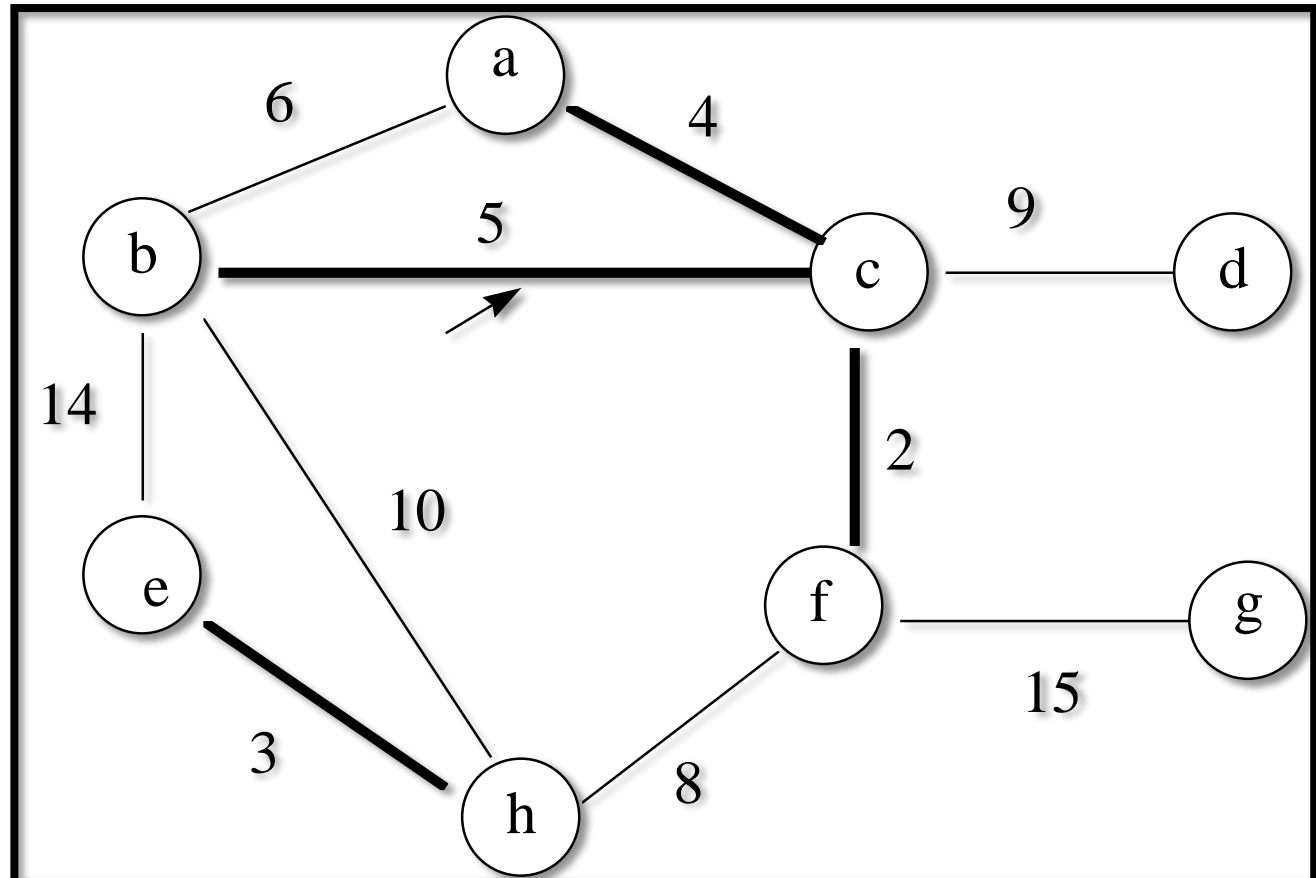


# Kruskal's Example

Keep going, checking next smallest edge.

Had  $\{b\}$   $\{a\ c\ f\}$   $\{d\}$   $\{e\ h\}$   $\{g\}$

$\{b\} \neq \{a\ c\ f\}$ , add edge.



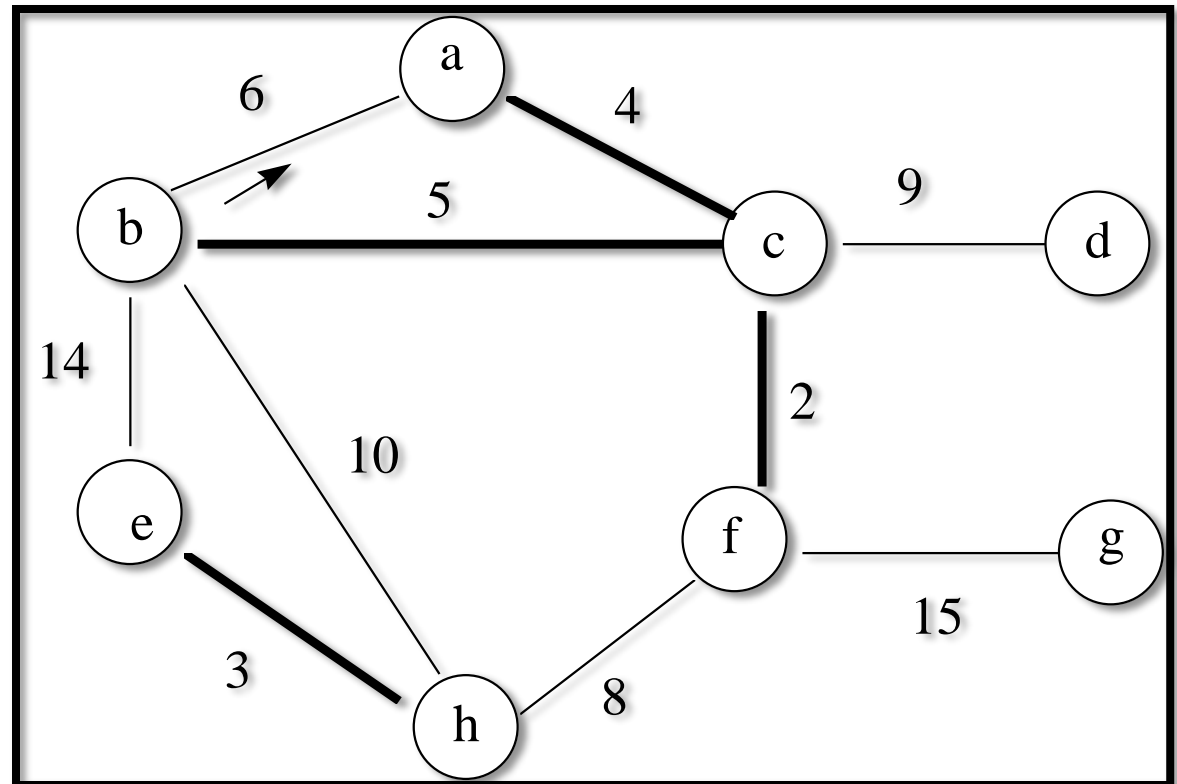
Now get  $\{a\ b\ c\ f\}$   
 $\{d\}$   $\{e\ h\}$   $\{g\}$

# Kruskal's Example

Keep going, checking next smallest edge.

Had  $\{a\ b\ c\ f\}$   $\{d\}$   $\{e\ h\}$   $\{g\}$

$\{a\ b\ c\ f\} = \{a\ b\ c\ f\}$ , don't add it!



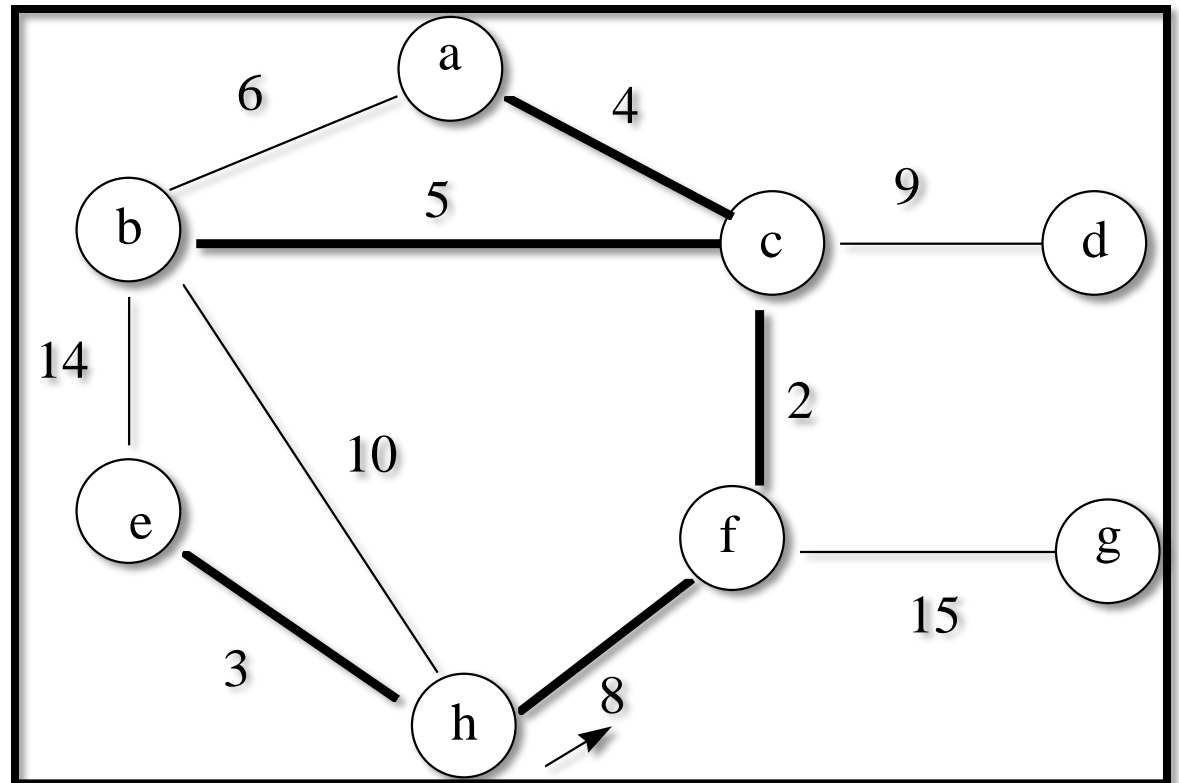
# Kruskal's Example

Keep going, checking next smallest edge.

Had  $\{a\ b\ c\ f\}$   $\{d\}$   $\{e\ h\}$   $\{g\}$

$\{a\ b\ c\ f\} = \{e\ h\}$ , add it.

Now get  
 $\{a\ b\ c\ f\ e\ h\}$   
 $\{d\}\{g\}$



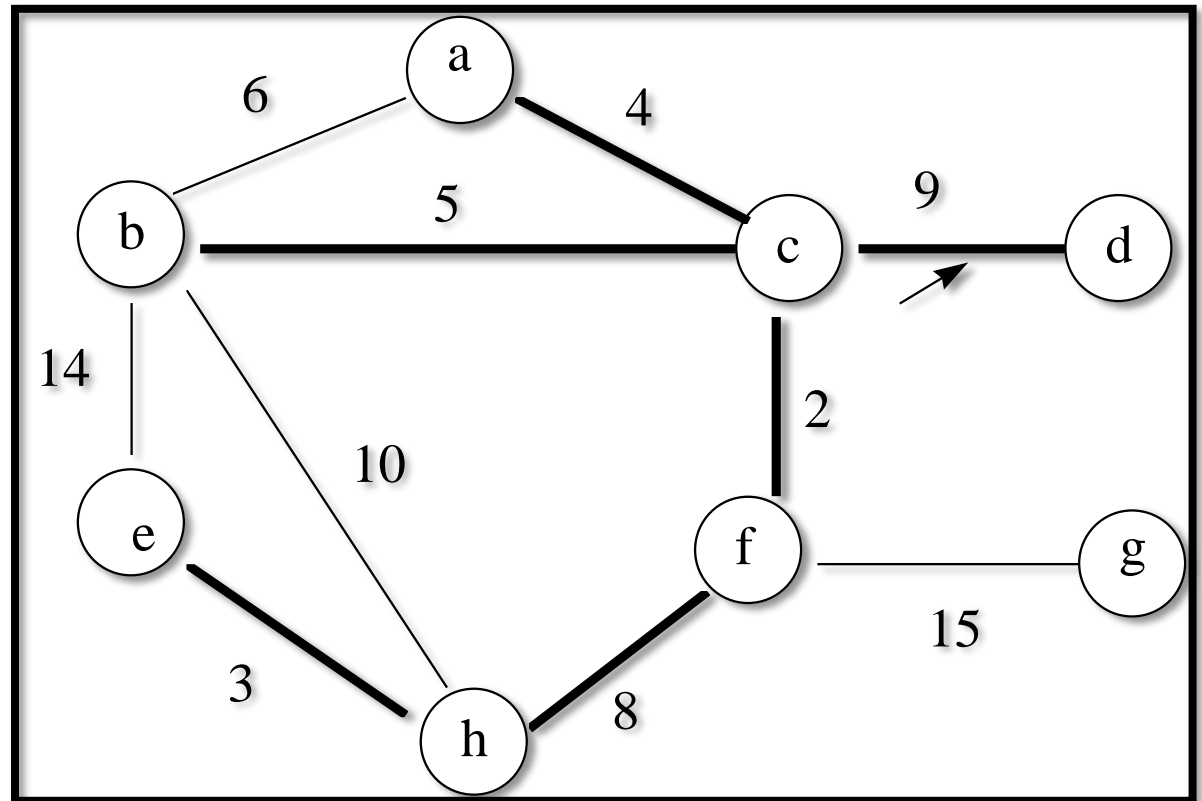
# Kruskal's Example

Keep going, checking next smallest edge.

Had  $\{a\ b\ c\ f\ e\ h\}\ \{d\}\{g\}$

$\{d\} \neq \{a\ b\ c\ e\ f\ h\}$ , add it.

Now get  
 $\{a\ b\ c\ d\ e\ f\ h\}$   
 $\{g\}$



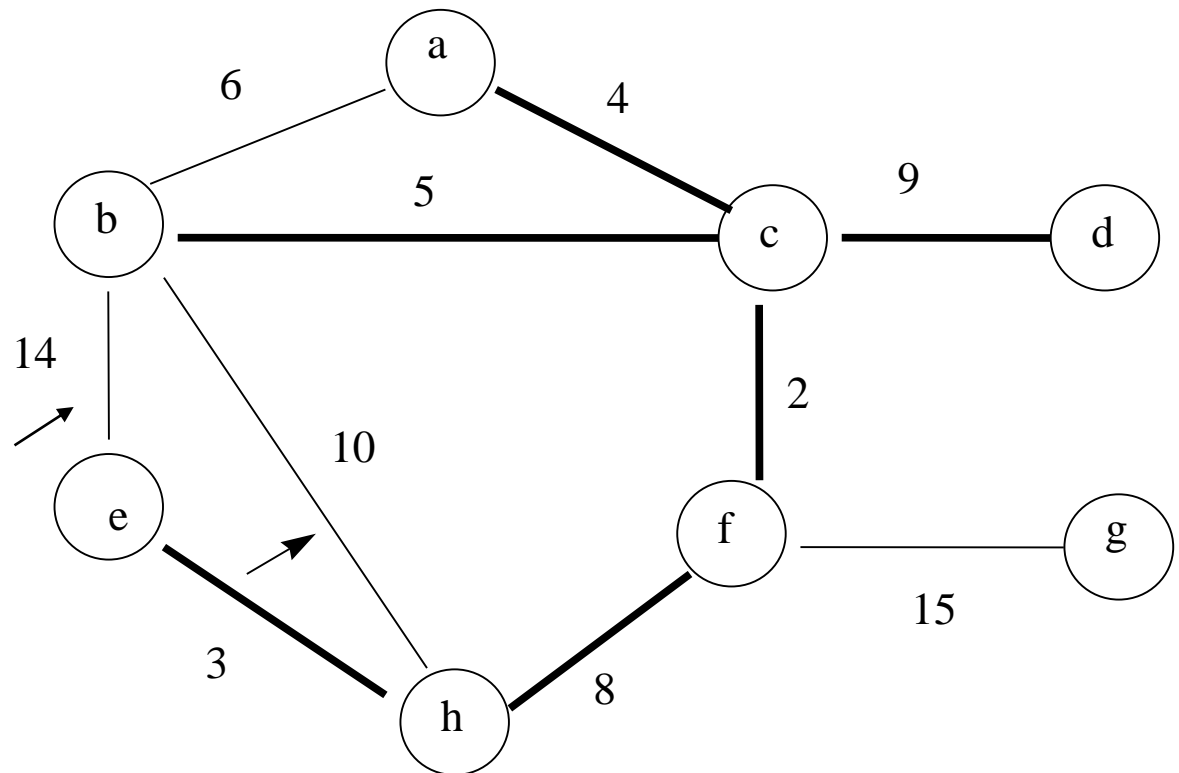


# Kruskal's Example

Keep going, check next two smallest edges.

Had  $\{a\ b\ c\ d\ e\ f\ h\}\ \{g\}$

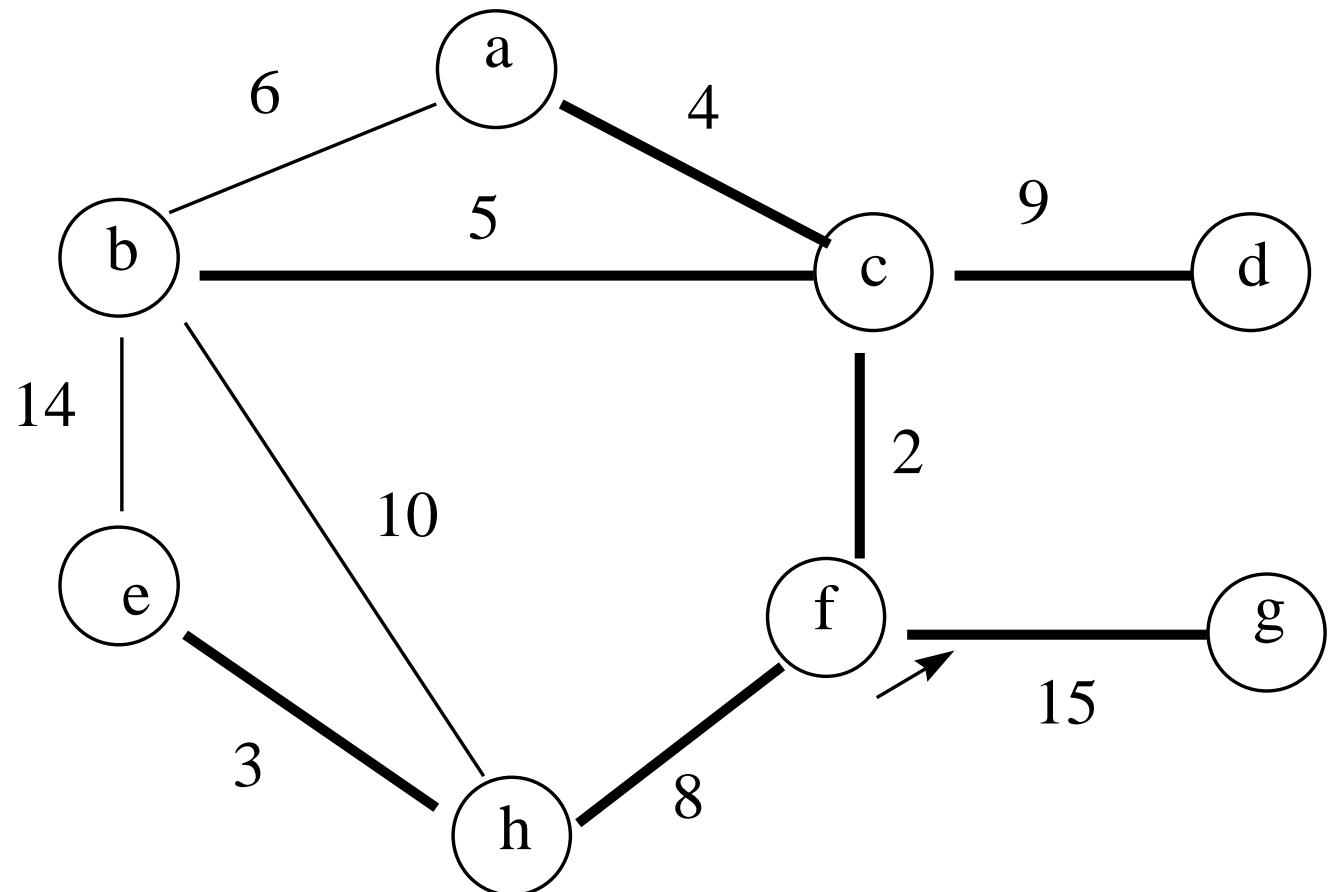
$\{a\ b\ c\ d\ e\ f\ h\} = \{a\ b\ c\ d\ e\ f\ h\}$ , don't add it.



# Kruskal's Example

Do add the last one:

Had  $\{a\ b\ c\ d\ e\ f\ h\}\ \{g\}$





# Runtime of Kruskal's Algo

---

- Runtime depends upon time to union set, find set, make set
- Simple set implementation: number each vertex and use an array
  - Use an array

$\text{member}[]$  :  $\text{member}[i]$  is a number  $j$  such that the  $i$ th vertex is a member of the  $j$ th set.
  - Example

$\text{member}[1,4,1,2,2]$   
indicates the sets  $S1=\{1,3\}$ ,  $S2=\{4,5\}$  and  $S4=\{2\}$ ;  
i.e. position in the array gives the set number. Idea similar to counting sort, up to number of edge members.

# Set Operations

- Given the Member array

- Make-Set( $v$ )

$\text{member}[v] = v$

Make-Set runs in constant running time for a single set.

- Find-Set( $v$ )

Return  $\text{member}[v]$

Find-Set runs in constant time.

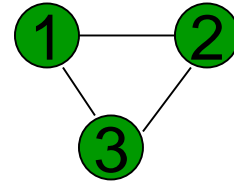
- Union( $u, v$ )

for  $i=1$  to  $n$

do if  $\text{member}[i] = u$  then  $\text{member}[i]=v$

Scan through the member array and update old members to be the new set.

Running time  $O(n)$ , length of member array.



$\text{member} =$   
 $[1, 2, 3] ; \{1\} \{2\} \{3\}$

$\text{find-set}(2) = 2$

Union(2,3)  
 $\text{member} = [1, 3, 3] ; \{1\}$   
 $\{2\} \{3\}$



# Overall Runtime

Kruskal( $G, w$ ) ; Graph  $G$ , with weights  $w$   $O(V)$   
     $A \leftarrow \{\}$  ; Our MST starts empty  
    for each vertex  $v \in V[G]$  do Make-Set( $v$ ) ; Make each vertex a set  
    Sort edges of  $E$  by increasing weight  $O(E \lg E)$  – using heapsort  
    for each edge  $(u, v) \in E$  in order  $O(E)$   
        ; Find-Set returns a representative (first vertex) in the set  
        do if Find-Set( $u$ )  $\neq$  Find-Set( $v$ )  $O(1)$   
            then  $A \leftarrow A \cup \{(u, v)\}$   
            Union( $u, v$ ) ; Combines two trees  $O(V)$   
    return  $A$

Total runtime:  $O(V) + O(E \lg E) + O(E * (1 + V)) = O(E * V)$

Book describes a version using disjoint sets that runs in  $O(E * \lg E)$  time