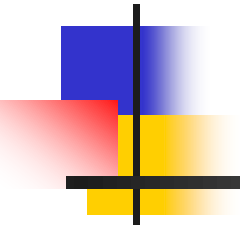
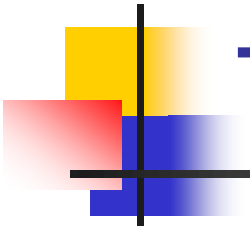
Algorithms and Data Structures
Lecture XII



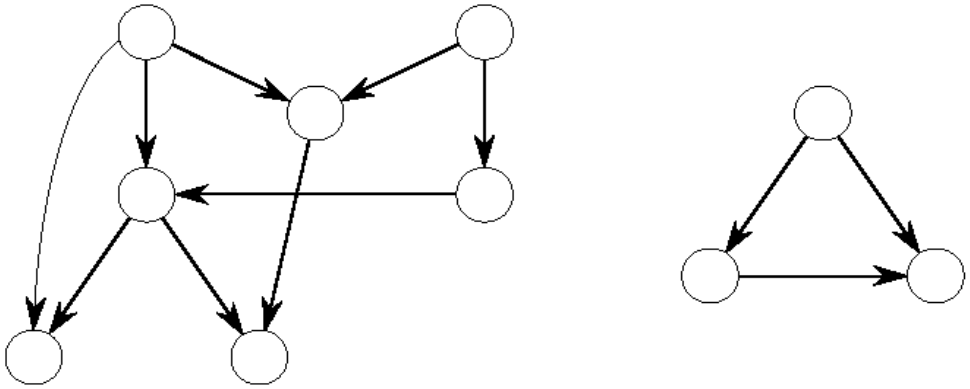


This Lecture

- Finish up Topological Sort
- Weighted Graphs
- Minimum Spanning Trees
  - Greedy Choice Theorem
  - Kruskal's Algorithm
  - Prim's Algorithm

Directed Acyclic Graphs

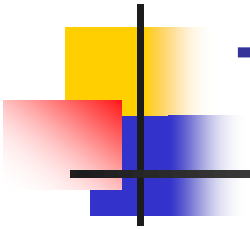
- A DAG is a directed graph with no cycles



- Often used to indicate precedences among events, i.e., event *a* must happen before *b*
- An example would be a parallel code execution
- Inducing a total order can be done using **Topological Sorting**

DAG Theorem

- A directed graph *G* is acyclic iff a DFS of *G* yields no back edges
- Proof
  - **suppose there is a back edge (*u*,*v*);** *v* is an ancestor of *u* in DFS forest. Thus, there is a path from *v* to *u* in G and (*u*,*v*) completes the cycle
  - **suppose there is a cycle *c*;** let *v* be the first vertex in *c* to be discovered and *u* is a predecessor of *v* in *c*.
    - Upon discovering *v* the whole cycle from *v* to *u* is white
    - We must visit all nodes reachable on this white path before return DFS-Visit(*v*), i.e., vertex *u* becomes a descendant of *v*
    - Thus, (*u*,*v*) is a back edge



Topological Sort

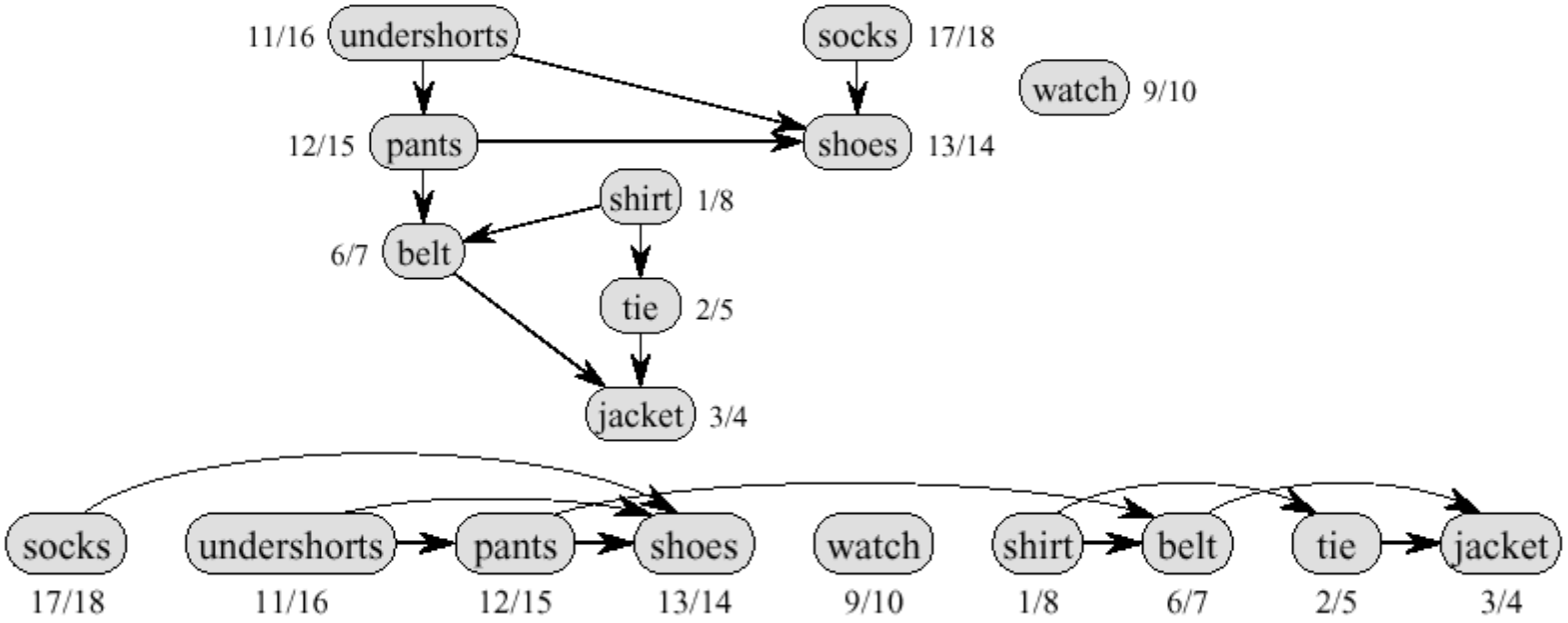
- Sorting of a directed acyclic graph (DAG)
- A topological sort of a DAG is a linear ordering of all its vertices sucht that for any edge (u,v) in the DAG, u appears before v in the ordering
- The following algorithm topologically sorts a DAG

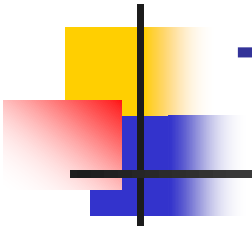
Topological-Sort(G)
1) call DFS(G) to compute finishing times $f[v]$ for each vertex v
2) as each vertex is finished, insert it onto the front of a linked list
3) return the linked list of vertices

- The linked lists comprises a total ordering

Topological Sort Example

- Precedence relations: an edge from x to y means one must be done with x before one can do y
- Intuition: can schedule task only when all of its subtasks have been scheduled





Topological Sort

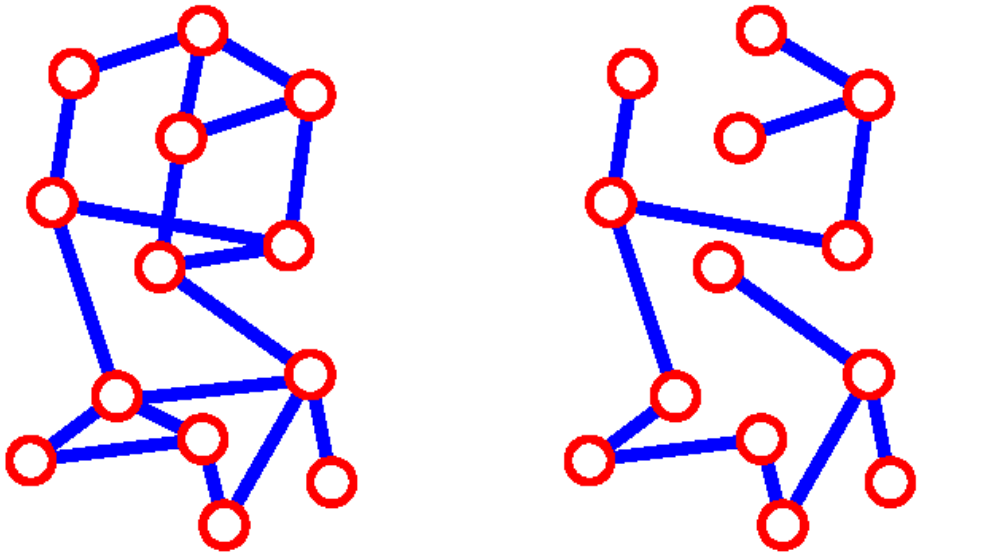
- Running time
  - depth-first search: $O(V+E)$ time
  - insert each of the $|V|$ vertices to the front of the linked list: $O(1)$ per insertion
- Thus the total running time is $O(V+E)$

Topological Sort Correctness

- Claim: for a DAG, an edge $(u,v) \in E \Rightarrow f[u] > f[v]$
- When (u,v) explored, u is gray. We can distinguish three cases
  - v = gray
    $\Rightarrow$ (u,v) = back edge (cycle, contradiction)
  - v = white
    $\Rightarrow$ v becomes descendant of u
    $\Rightarrow$ v will be finished before u
    $\Rightarrow$ f[v] < f[u]
  - v = black
    $\Rightarrow$ v is already finished
    $\Rightarrow$ f[v] < f[u]
- The definition of topological sort is satisfied

Spanning Tree

- A **spanning tree** of **G** is a subgraph which
  - is a tree
  - contains all vertices of **G**

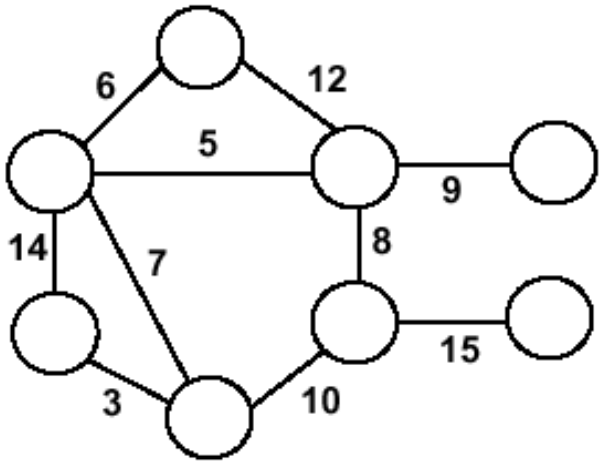


G

spanning tree of G

Minimum Spanning Trees

- Undirected, connected graph $G = (V, E)$
- Weight function W: $E \rightarrow R$ (assigning cost or length or other values to edges)

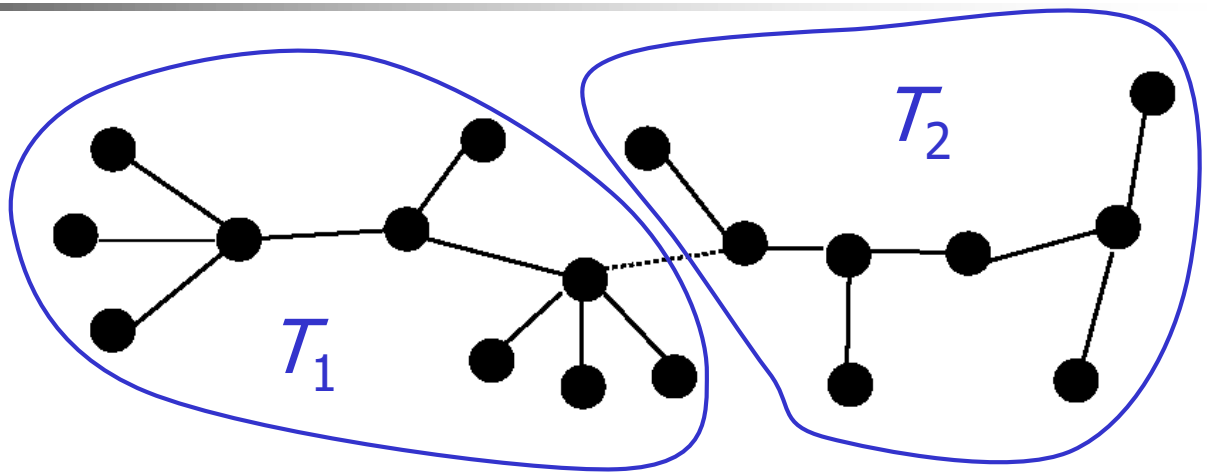


- Spanning tree: tree that connects all the vertices (above?)
- Minimum spanning tree: tree that connects all the vertices and minimizes

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

Optimal Substructure

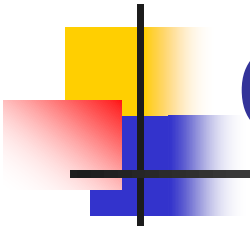
- MST T



- Removing the edge (u,v) partitions T into T_1 and T_2

$$w(T) = w(u,v) + w(T_1) + w(T_2)$$

- We claim that T_1 is the MST of $G_1=(V_1,E_1)$, the subgraph of G induced by vertices in T_1
- Also, T_2 is the MST of G_2

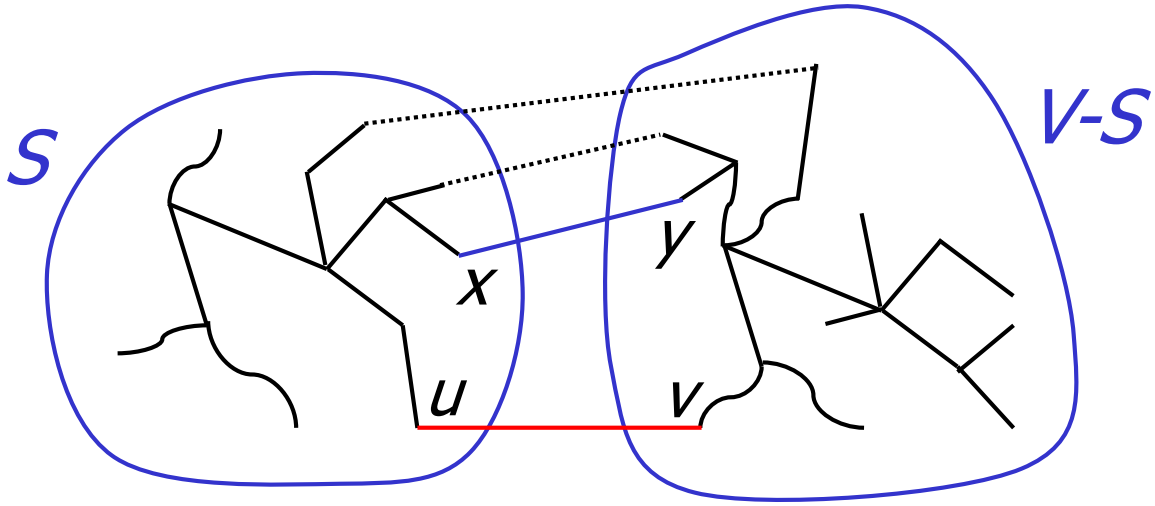


Greedy Choice

- Greedy choice property: locally optimal (greedy) choice yields a globally optimal solution
- Theorem
  - Let $G=(V, E)$, and let $S \subseteq V$ and
  - let (u,v) be min-weight edge in G connecting S to $V - S$
  - Then $(u,v) \in T$ – some MST of G

Greedy Choice (2)

- Proof
  - suppose $(u,v) \notin T$
  - look at path from u to v in T
  - swap (x, y) – the first edge on path from u to v in T that crosses from S to $V - S$
  - this improves T – contradiction (T supposed to be MST)



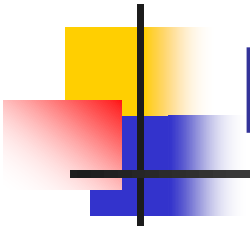
Generic MST Algorithm

```
Generic-MST(G, w)
1 A←∅  // 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∪{(u,v)}
5 return A
```

Safe edge – edge that does not destroy *A*'s property

```
MoreSpecific-MST(G, w)
1    A←∅  // 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∪{(u,v)}
5 return A
```

Prim-Jarnik Algorithm



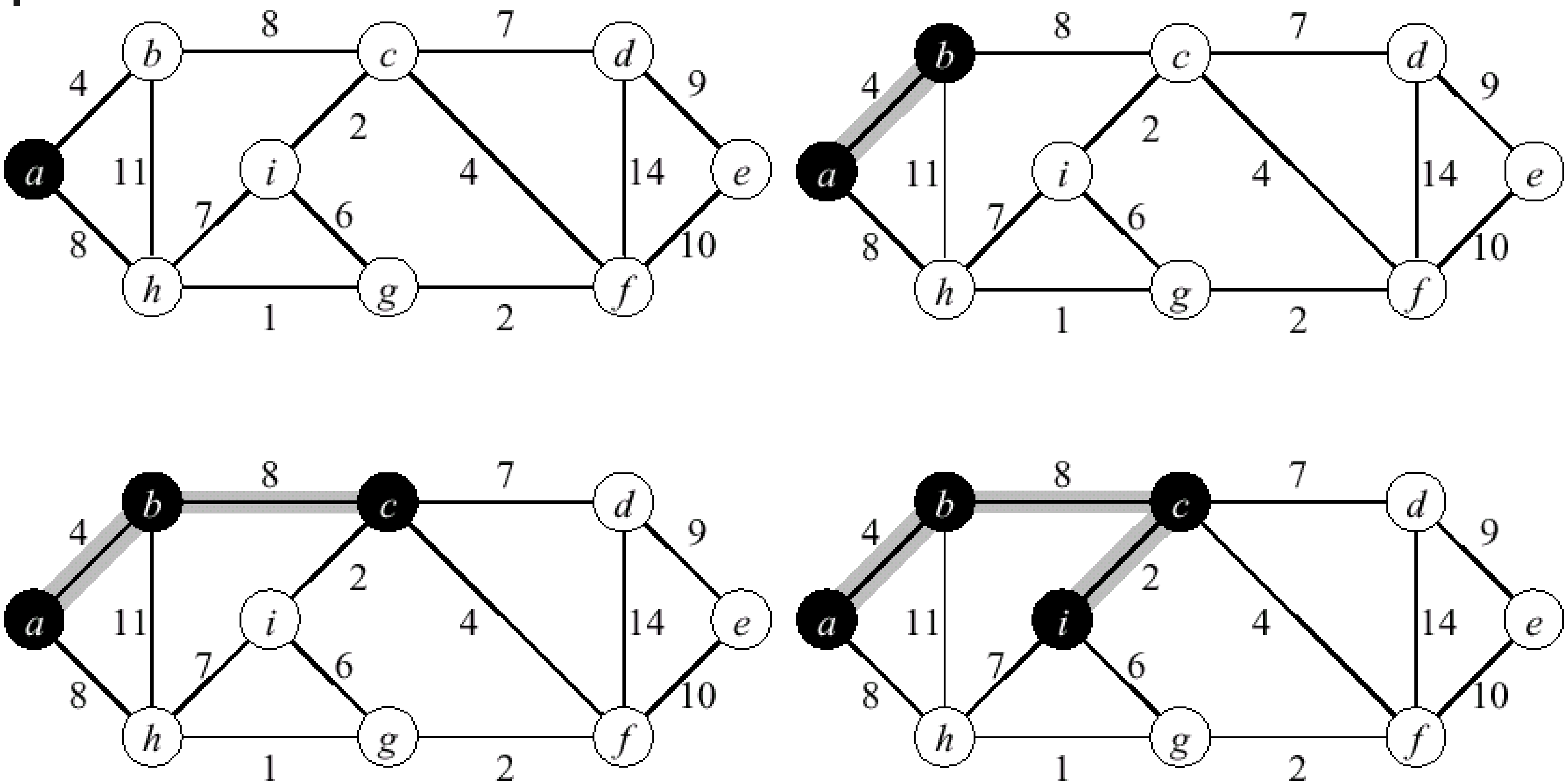
- Vertex based algorithm
- Grows one tree T, **one vertex at a time**
- A cloud covering the portion of T already computed
- Label the vertices v outside the cloud with $key[v]$ – the minimum weigth of an edge connecting v to a vertex in the cloud, $key[v] = \infty$, if no such edge exists

Prim-Jarnik Algorithm (2)

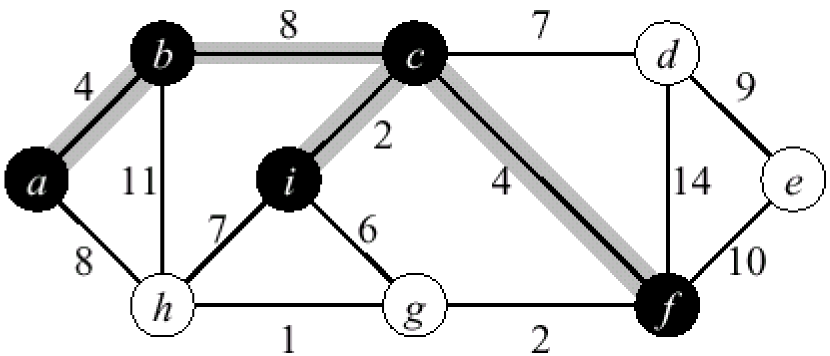
```
MST-Prim(G,w,r)
01 Q ← V[G]  // Q - vertices out of T
02 for each u ∈ Q
03    key[u] ← ∞
04 key[r] ← 0
05 π[r] ← NIL
06 while Q ≠ ∅ do
07   u ← ExtractMin(Q)  // making u part of T
08      for each v ∈ Adj[u] do
09          if v ∈ Q and w(u,v) < key[v] then
10             π[v] ← u
11             key[v] ← w(u,v)
```

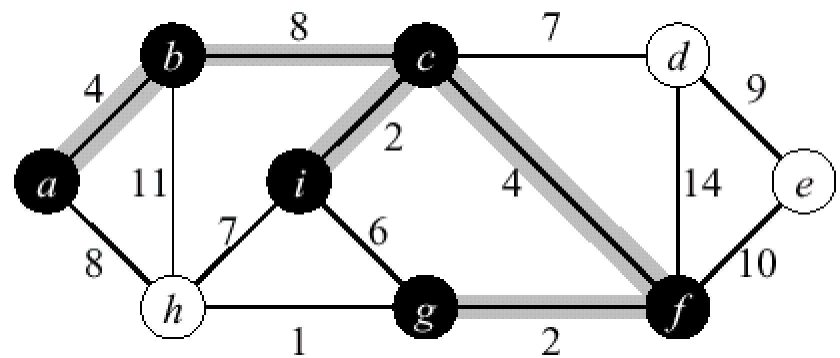
updating keys

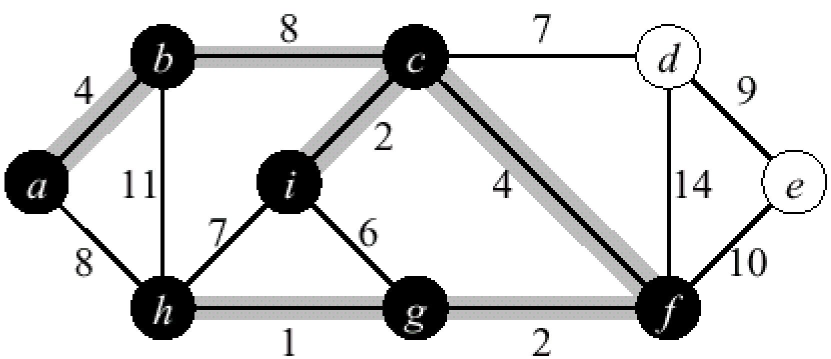
Prim Example

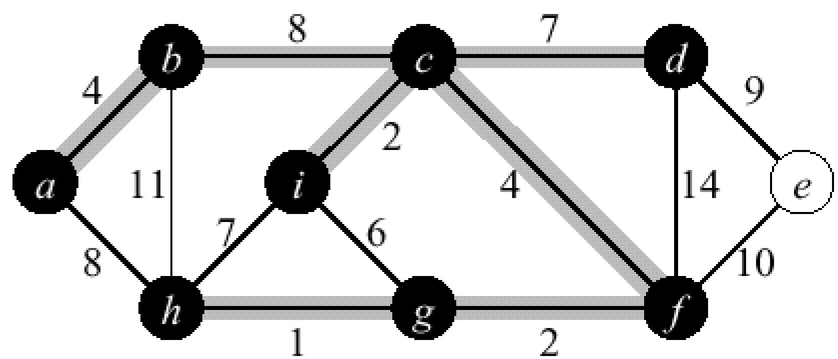


Prim Example (2)

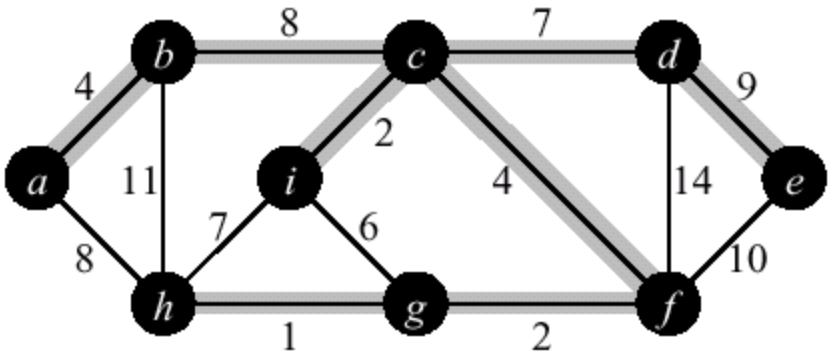


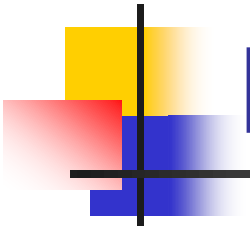






Prim Example (3)





Priority Queues

- A priority queue is a data structure for maintaining a set S of elements, each with an associated value called key
- We need PQ to support the following operations
  - BuildPQ(S) – initializes PQ to contain elements of S
  - ExtractMin(S) returns and removes the element of S with the smallest key
  - ModifyKey(S,x,newkey) – changes the key of x in S
- A binary heap can be used to implement a PQ
  - BuildPQ – $O(n)$
  - ExtractMin and ModifyKey – $O(\lg n)$

Prim's Running Time

- Time = $|V| T(\text{ExtractMin}) + O(E) T(\text{ModifyKey})$
- Time = $O(V \lg V + E \lg V) = O(E \lg V)$

Q	T(ExtractMin)	T(DecreaseKey)	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$	$O(1)$ amortized	$O(V \lg V + E)$

Kruskal's Algorithm

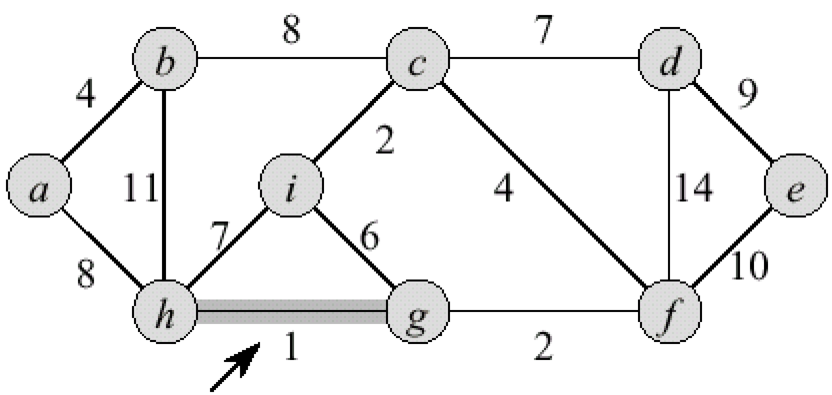
- Edge based algorithm
- Add the edges one at a time, in increasing weight order
- The algorithm maintains A – a **forest of trees**. An edge is accepted it if connects vertices of distinct trees
- We need a data structure that maintains a partition, i.e.,a collection of disjoint sets
  - MakeSet(S,x): $S \leftarrow S \cup \{\{x\}\}$
  - Union(S_i,S_j): $S \leftarrow S - \{S_i,S_j\} \cup \{S_i \cup S_j\}$
  - FindSet(S, x): returns unique $S_i \in S$, where $x \in S_i$

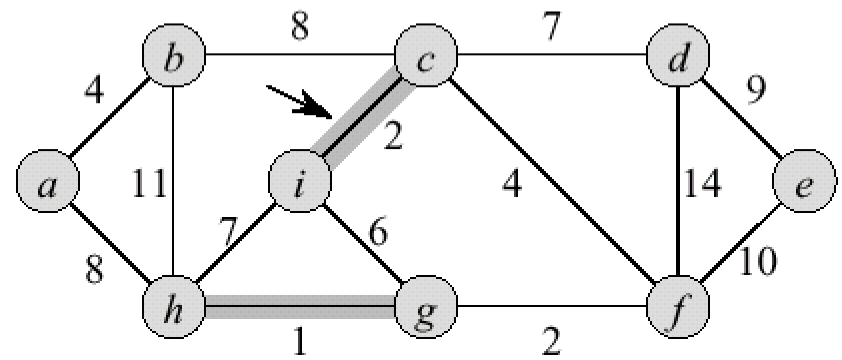
Kruskal's Algorithm

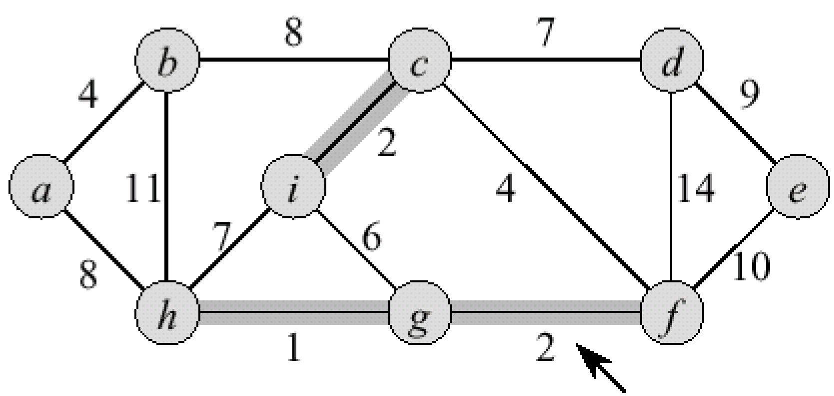
- The algorithm adds the cheapest edge that connects two trees of the forest

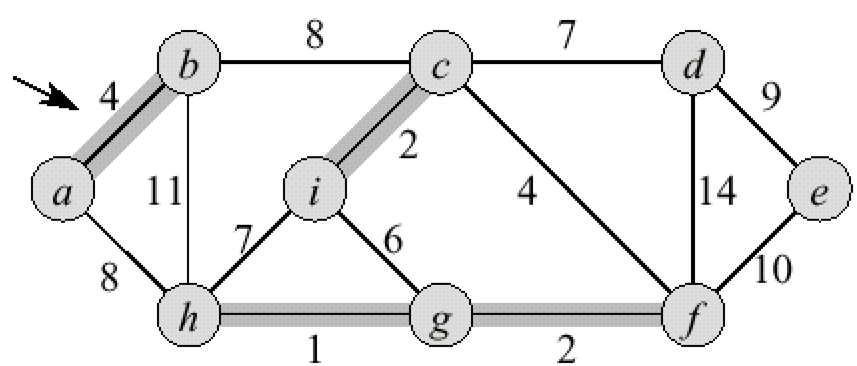
```
MST-Kruskal(G,w)
01 A ← ∅
02 for each vertex v ∈ V[G] do
03    Make-Set(v)
04 sort the edges of E by non-decreasing weight w
05  for each edge (u,v) ∈ E, in order by non-
   decreasing weight do
06    if Find-Set(u) ≠ Find-Set(v) then
07       A ← A ∪ {(u,v)}
08       Union(u,v)
09 return A
```

Kruskal Example

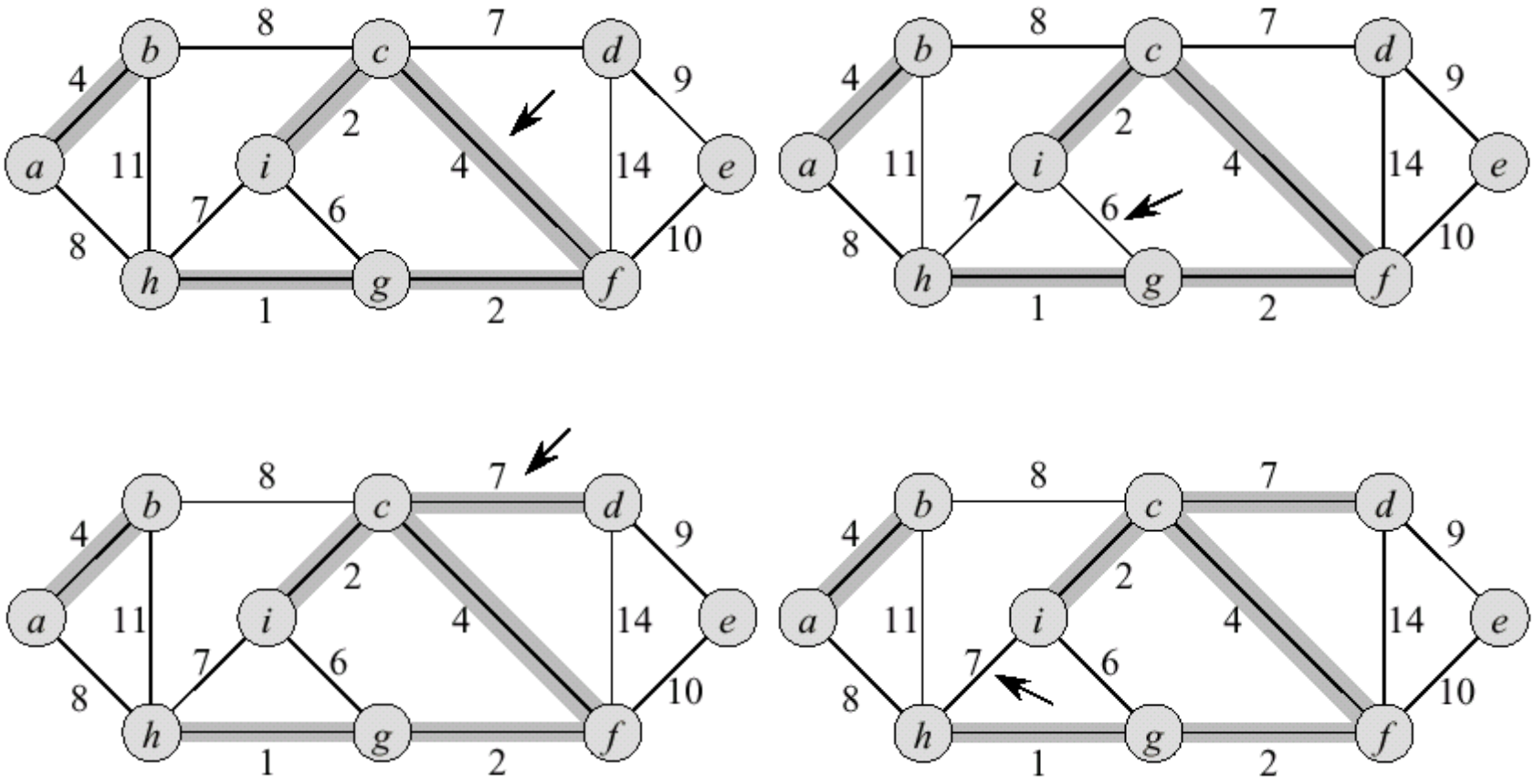




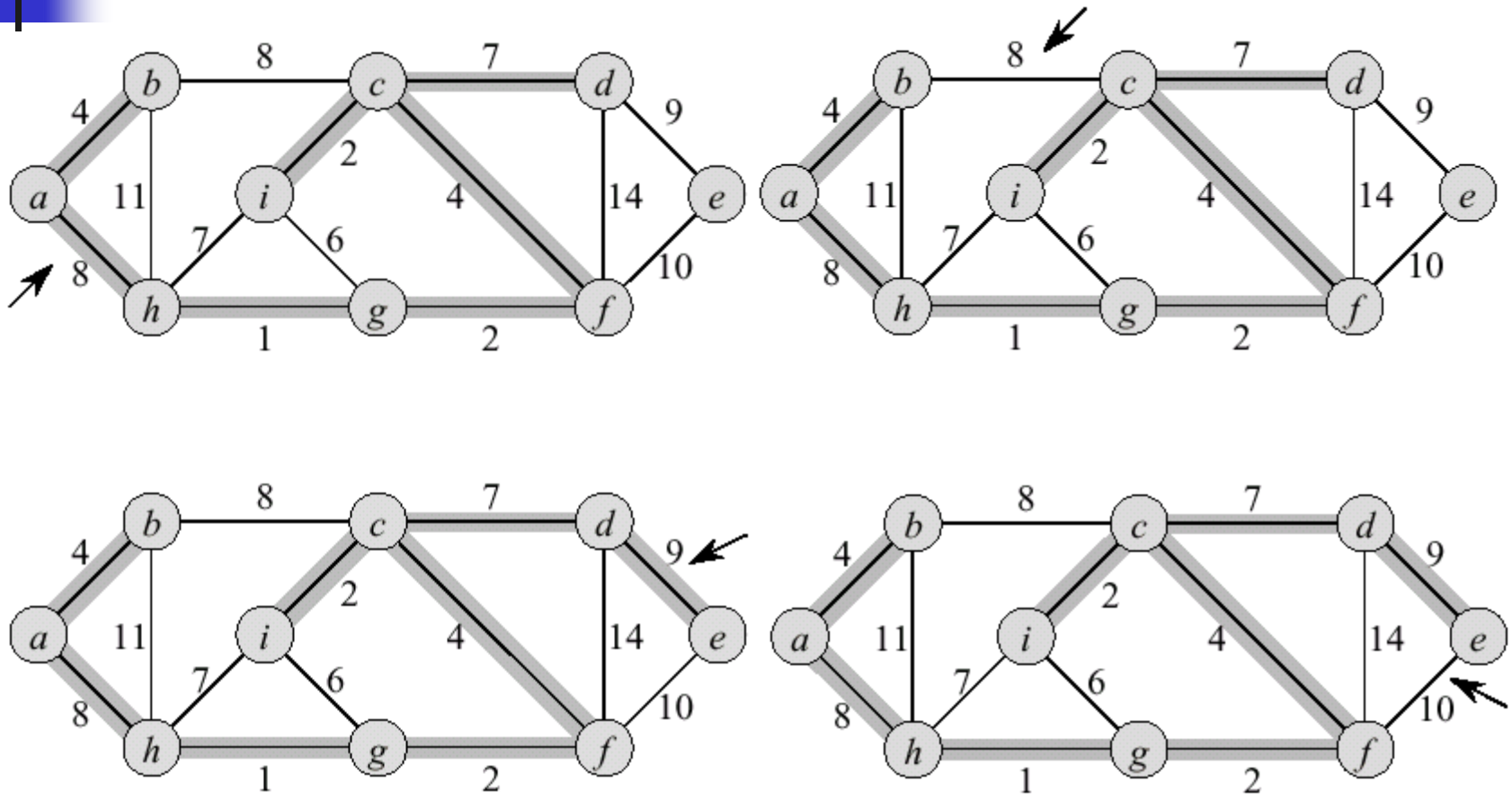




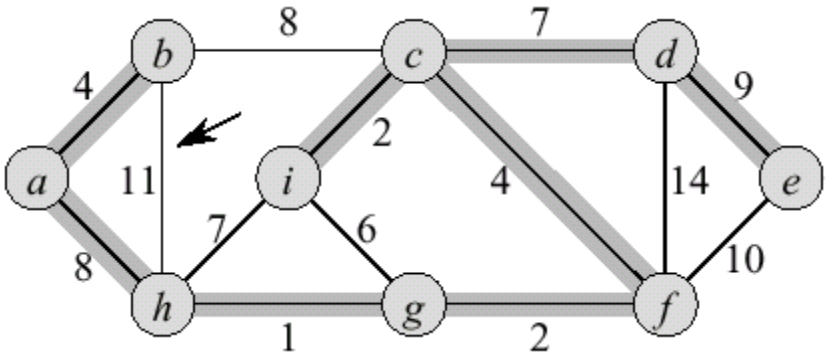
Kruskal Example (2)

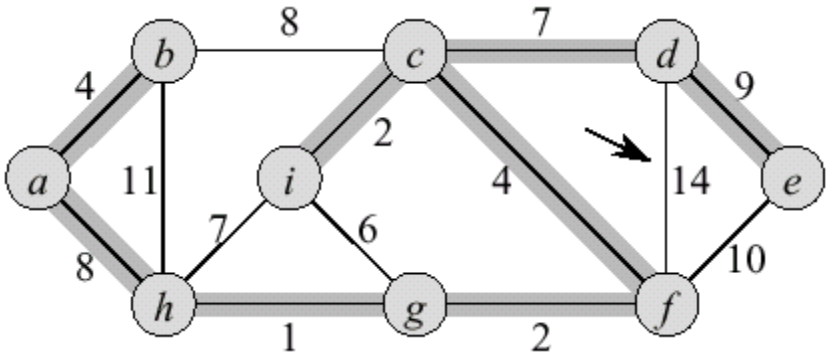


Kruskal Example (3)



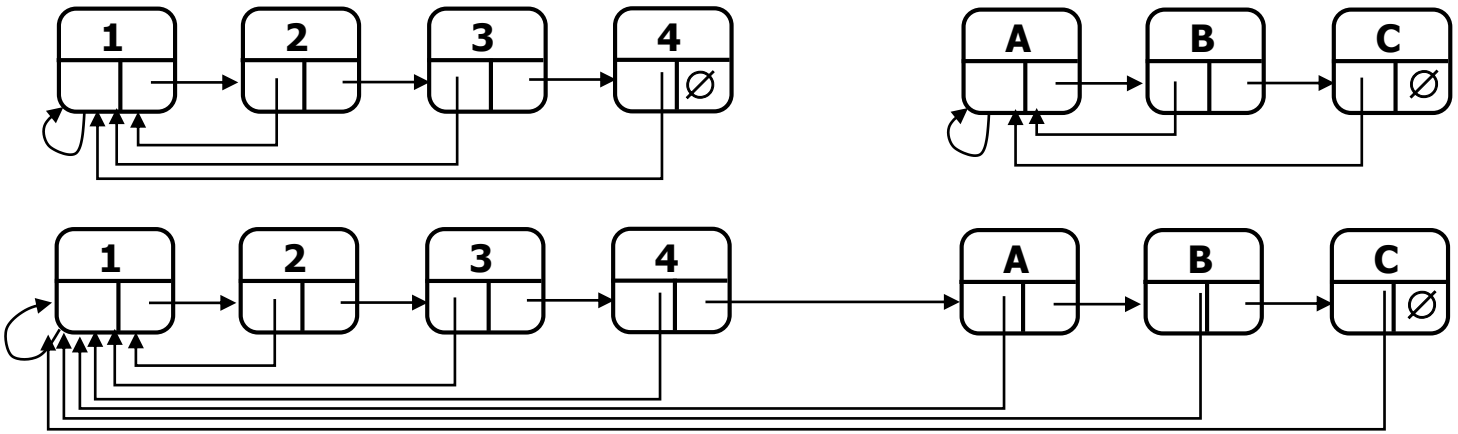
Kruskal Example (4)





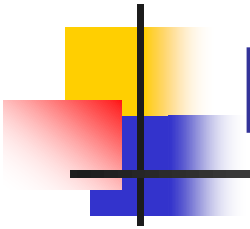
Disjoint Sets as Lists

- Each set – a list of elements identified by the first element, all elements in the list point to the first element
- Union – add a smaller list to a larger one
- FindSet: $O(1)$, Union(u,v): $O(\min\{|C(u)|, C(v)\})$



Kruskal Running Time

- Initialization $O(V)$ time
- Sorting the edges $\Theta(E \lg E) = \Theta(E \lg V)$ (why?)
- $O(E)$ calls to FindSet
- Union costs
  - Let $t(v)$ – 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: $O(E \lg V)$



Next lecture

- Shortest Paths in Weighted Graphs