CISC 235: Topic 10

Graph Algorithms

Outline

- Spanning Trees
- Minimum Spanning Trees
  - Kruskal's Algorithm
  - Prim's Algorithm
- Topological Sort

Spanning Trees

A *spanning tree* of an undirected graph is a subgraph that contains all its vertices and is a tree.

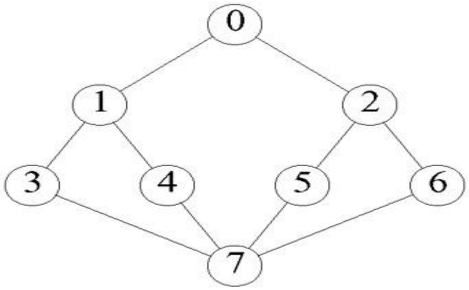
The complete graph on four vertices

```
o---o
|\ /|
| x |
|/ \|
o---o
```

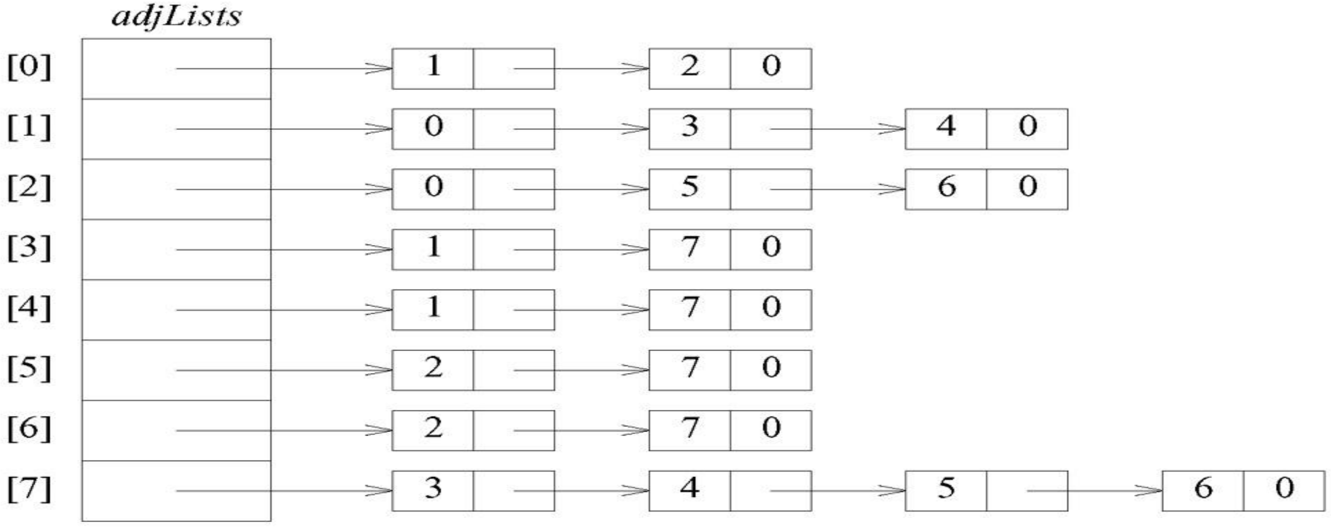
has 16 spanning trees.

```
o---o    o---o    o   o    o---o
|   |    |        |   |        |
|   |    |        |   |        |
|   |    |        |   |        |
o   o    o---o    o---o    o---o

o---o    o   o    o   o    o   o
 \ /     |\ /      \ /      \ /|
  x      | x        x        x |
 / \     |/ \      / \      / \|
o   o    o   o    o---o    o   o

o   o    o---o    o   o    o---o
|\  |       /     |  /|     \
| \ |      /      | / |      \
|  \|     /       |/  |       \
o   o    o---o    o   o    o---o

o---o    o   o    o   o    o---o
|\       |  /      \  |       /|
| \      | /        \ |      / |
|  \     |/          \|     /  |
o   o    o---o    o---o    o   o
```

Example Graph



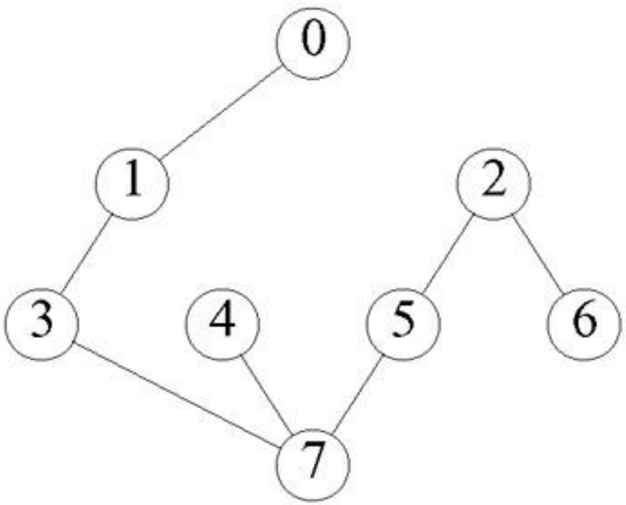
(a)



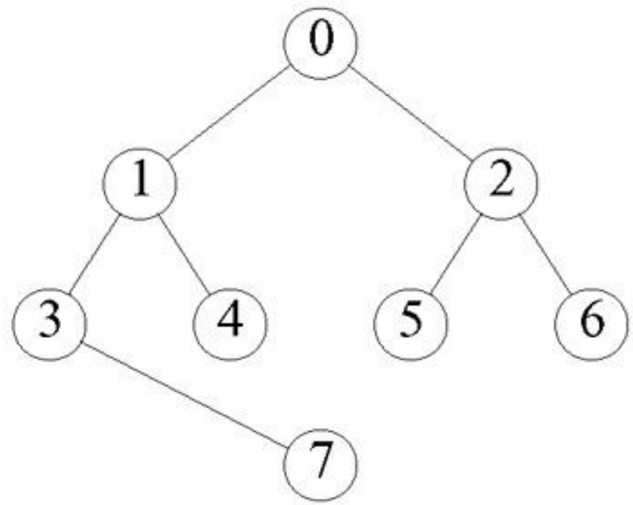
(b)

DFS & BFS Spanning Trees

What property does the BFS spanning tree have that the DFS spanning tree does not?



(a) *DFS* (0) spanning tree

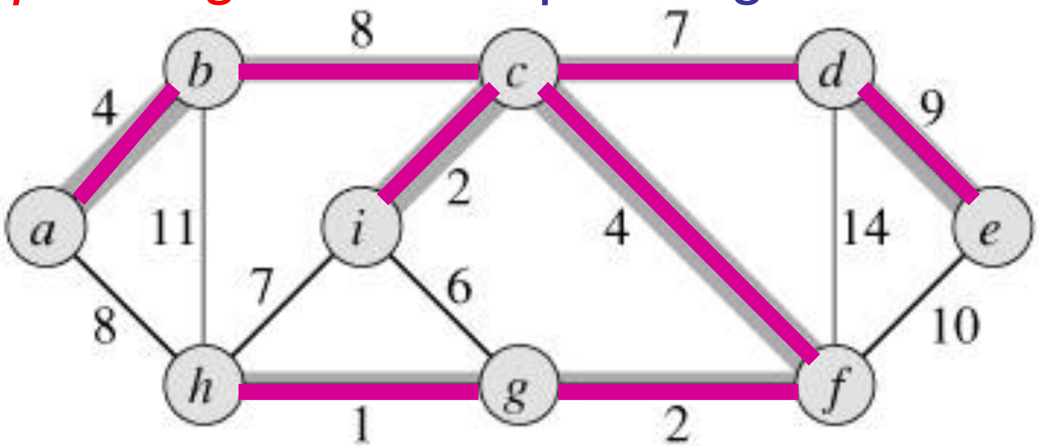


(b) *BFS* (0) spanning tree

Minimum Spanning Trees

The *cost* of a spanning tree of a weighted, undirected graph is the sum of the costs (weights) of the edges in the spanning tree.

A *minimum-cost spanning tree* is a spanning tree of least cost.



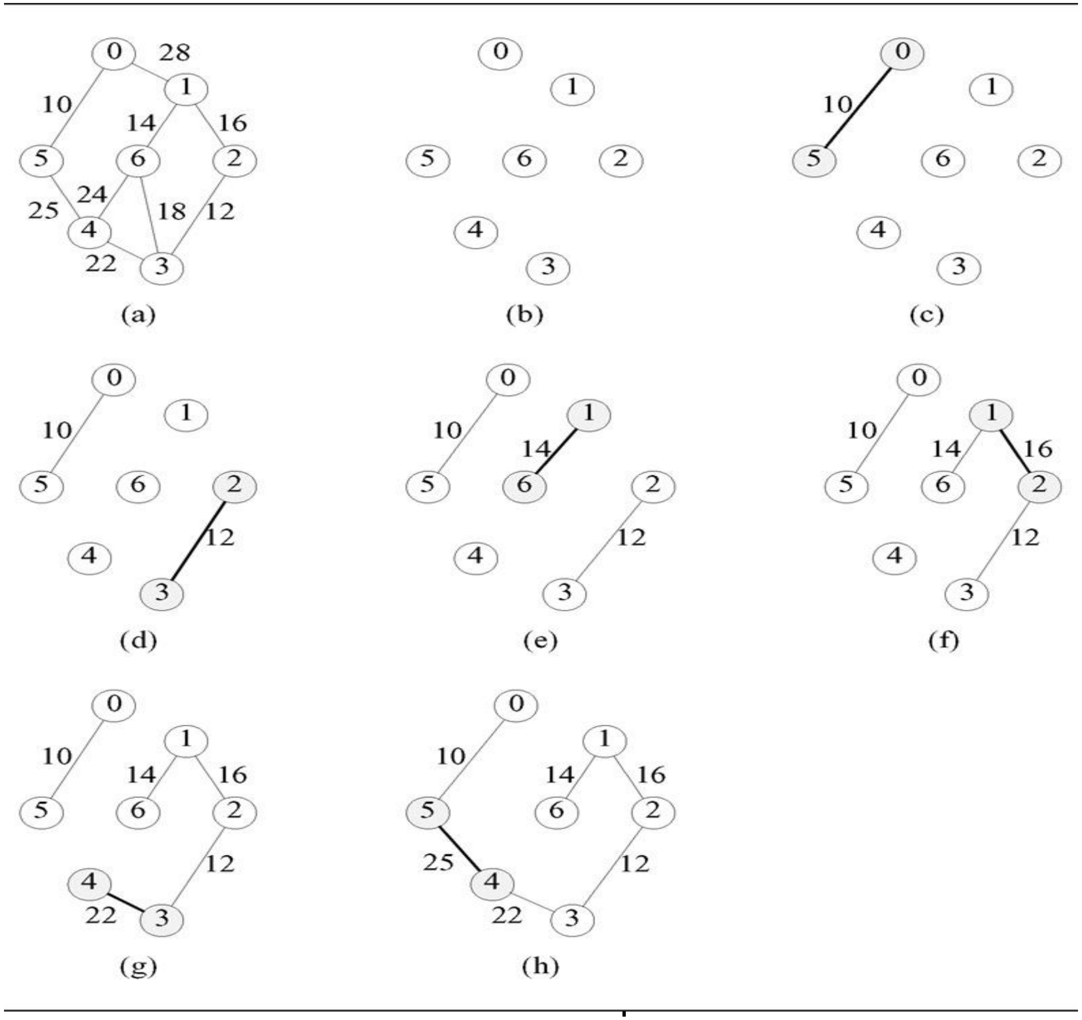
Is there more than one minimum spanning tree?

Kruskal's Algorithm

Begin with a subgraph *S* of a weighted, connected undirected graph *G*, that includes all of its vertices, but none of its edges (i.e., a forest of one-node trees).

Add edges from *G* one at a time to *S*, each time picking a least-cost edge that will not form a cycle in *S*, until *V*-1 edges have been added (so we have a single spanning tree).

Kruskal's Algorithm



Kruskal's Algorithm: $O(|E| \lg |V|)$

MST-Kruskal(G, w)   // weight function $w : E \rightarrow \mathbf{R}$

S ← Ø   // Subgraph S has no edges (but has all vertices)

for each vertex v ∈ V[G]

    MAKE-SET(v) // Each vertex is a tree of one node

sort the edges of E into nondecreasing order by weight w

for each edge (u, v) ∈ E, taken in nondecreasing order

    if FIND-SET(u) ≠ FIND-SET(v) // If not in same tree

        S ← S ∪ {(u, v)} // Add edge to subgraph S

        UNION(u, v) // Join their two trees

return S

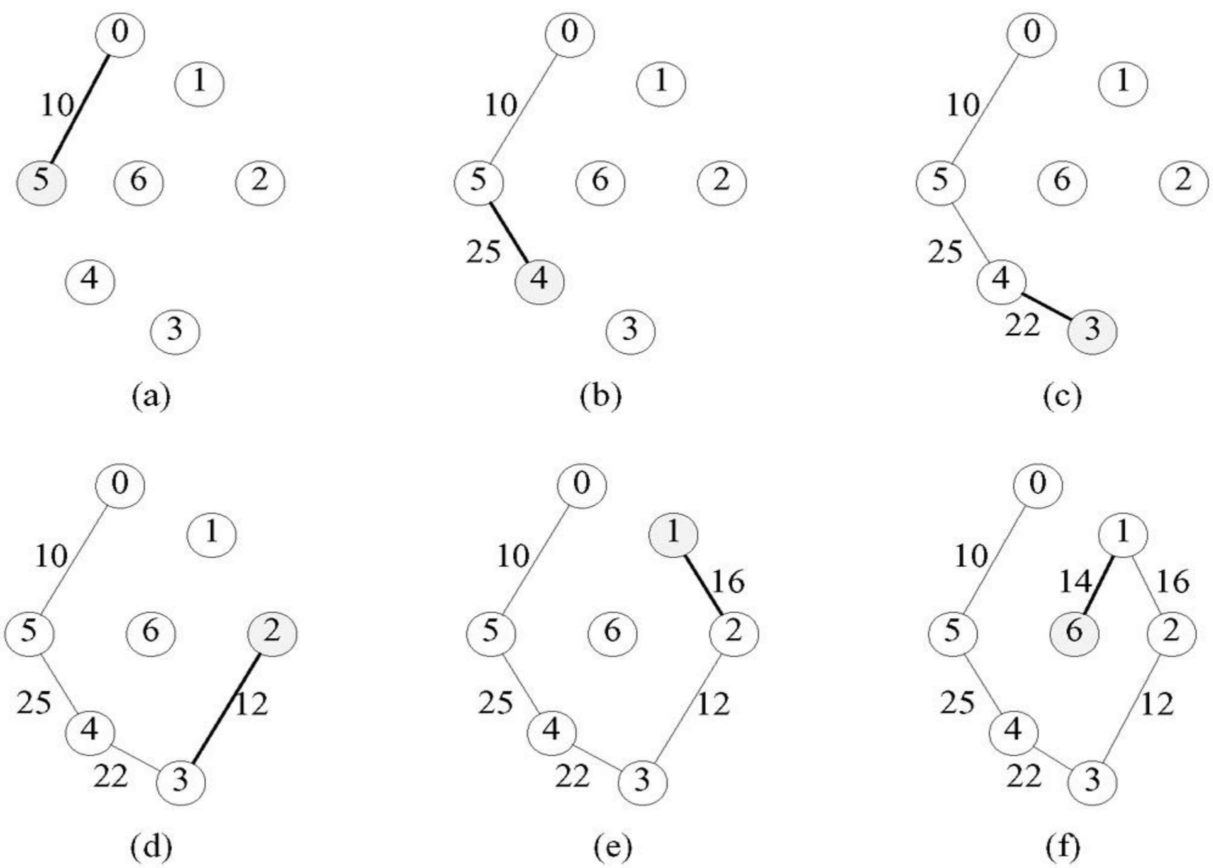
Prim's Algorithm

Begin with a tree *T* that contains any single vertex, *w*, from a weighted, connected undirected graph *G*.

Then add a least-cost edge (u, w) to *T* such that $T \cup \{(u, w)\}$ is also a tree. This edge-addition step is repeated until *T* contains $V\text{-}1$ edges.

Notice that edge (u, w) is always such that exactly one of *u* and *w* is in *T*.

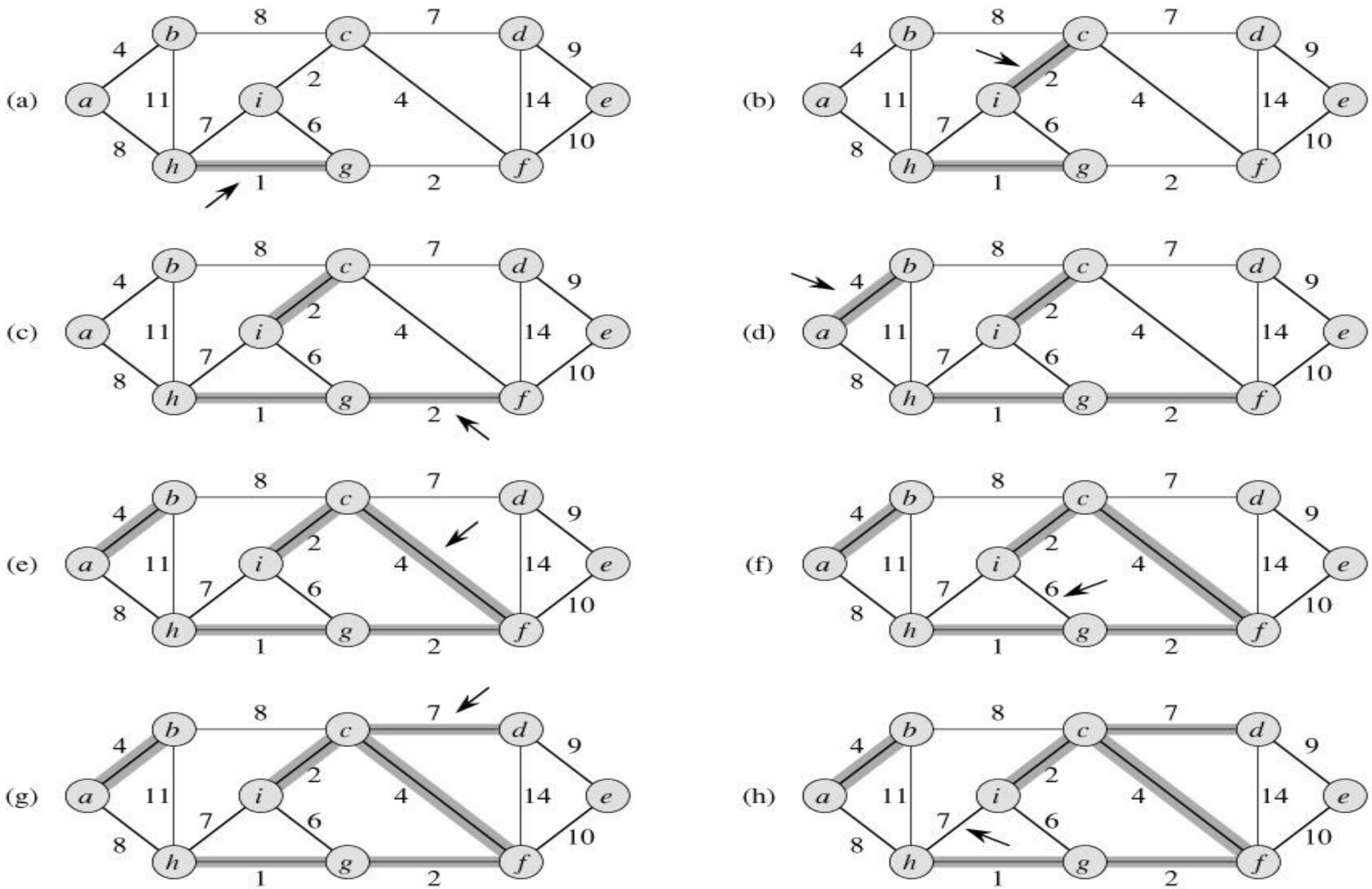
Prim’s Algorithm



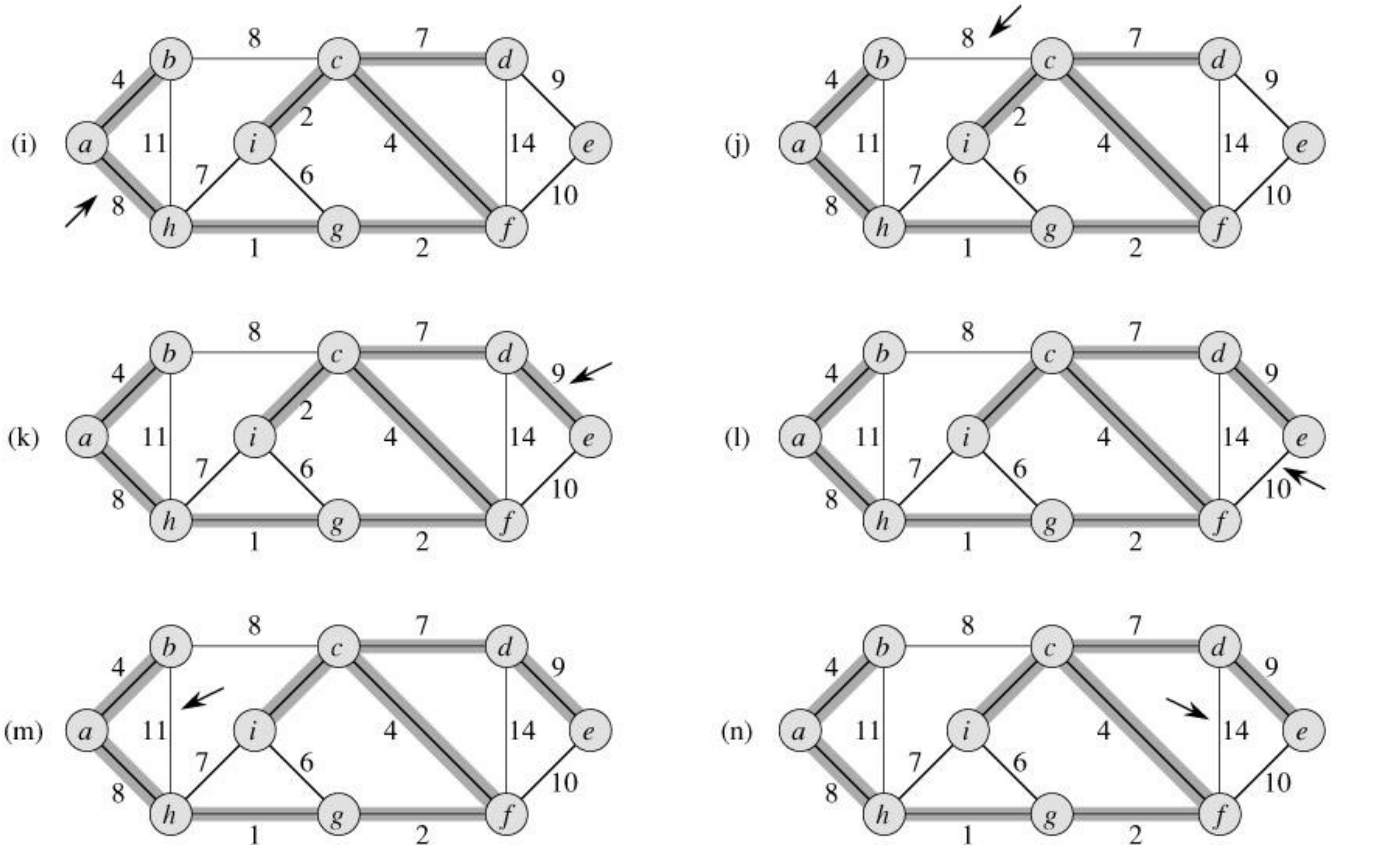
Prim's Algorithm Using a Priority Queue : O(|E| lg |V|)

```
MST-Prim(G, w, r)  // r is the root vertex (start vertex)
    for each u ∈ V [G]  // Initially, all vertices u are set to:
        key[u] ← ∞   // No edge connecting u to MST, so cost is ∞
        π[u] ← NIL   // No parent of u in MST
    key[r] ← 0  // Initially, r is the only vertex in MST
    Q ← V [G]  // All vertices are placed in priority queue
    while Q ≠ Ø
        u ← EXTRACT-MIN(Q)
        for each v ∈ Adj[u]  // Update adjacent edges if
            if v ∈ Q and w(u, v) < key[v]  // this path is cheaper
                π[v] ← u  // Reset parent of v to be u
                key[v] ← w(u, v)  // Reset minimum cost
```

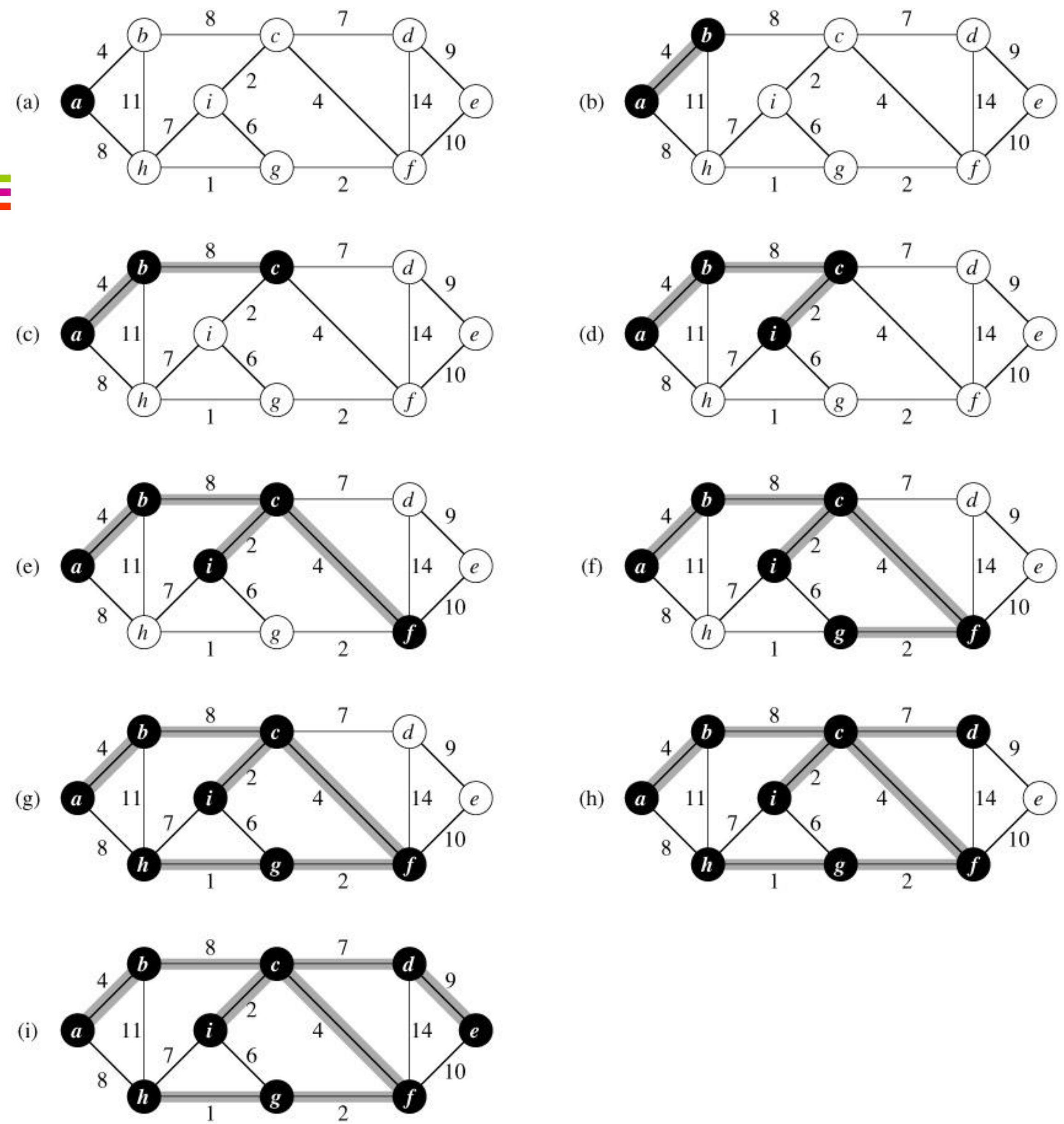
Kruskal’s Algorithm



Kruskal’s Algorithm, con.



Prim's Algorithm



Topological Sorting

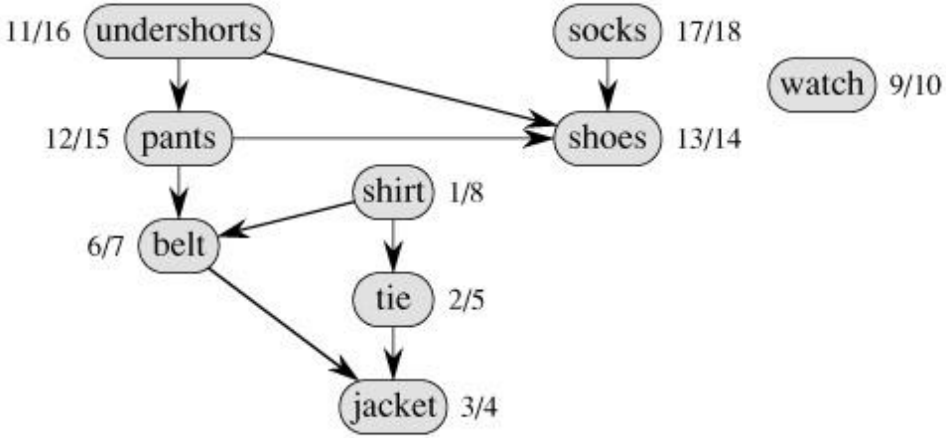
A *topological sort* of a *dag* (directed, acyclic graph) is a linear ordering of all its vertices such that if the graph contains an edge (u, v), then u appears before v in the ordering.

A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.

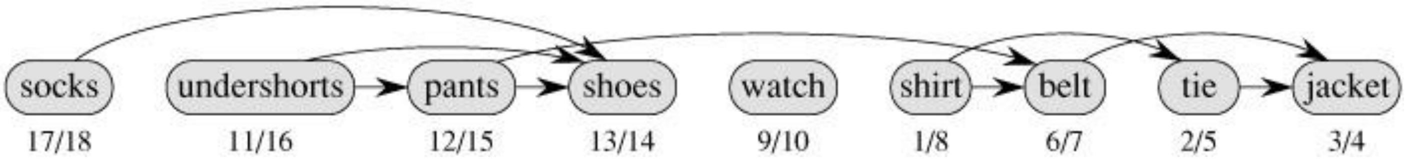
If the graph has a cycle, then no linear ordering is possible.

Topological Sorting

(a)

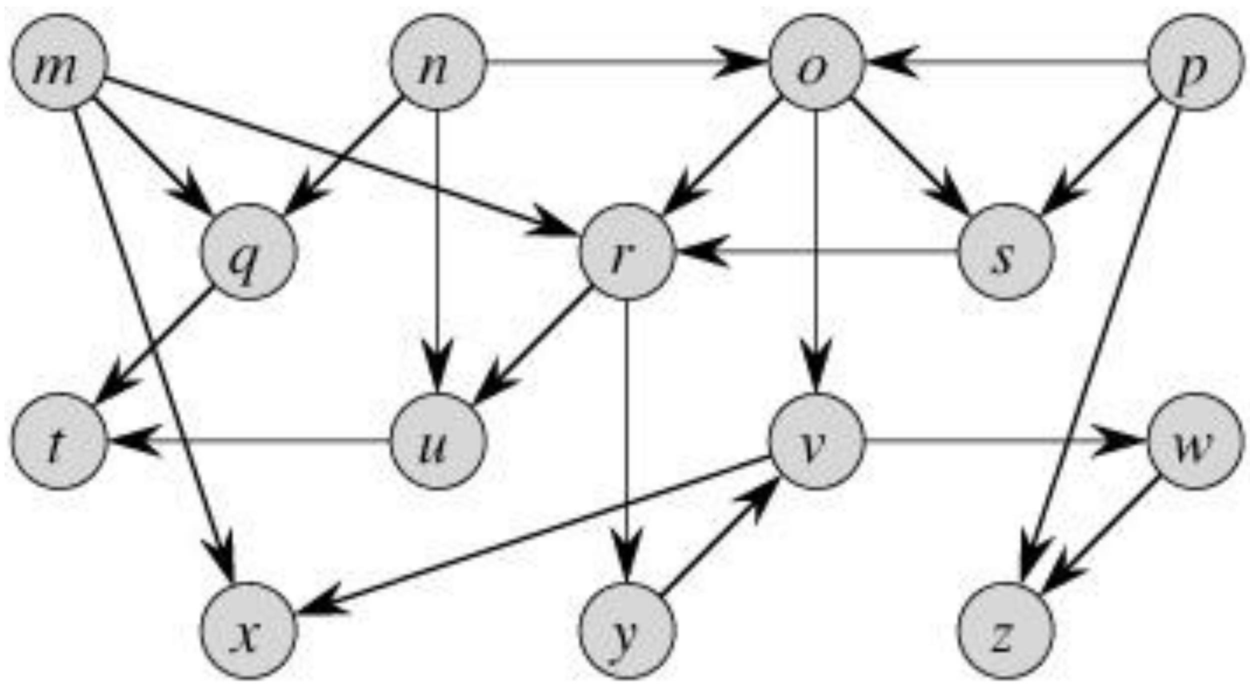


(b)



What other topological sorts are possible?

A *dag* for Topological Sorting



In general, which vertex will be first in a topological sort?

Topological Sort Algorithm

Begin with an empty list *L* and a dag *G*.

While *G* is not empty,

- Find a vertex *v* with no incoming edges
- Delete *v* and its outgoing edges from *G*
- Add *v* to *L*

Topological Sort Algorithm

```
Topological-Sort( G )
    Create empty lists L & K
    Create a count array
    for each vertex v in G
        count[v] ← number of incoming edges to v
        if count[v] = 0
            add v to K
    while K is not empty
        remove a vertex v from K
        for each outgoing edge (v,w)
            decrement count[w]
            if count[w] = 0
                add w to K
        add v to L
    return L
```