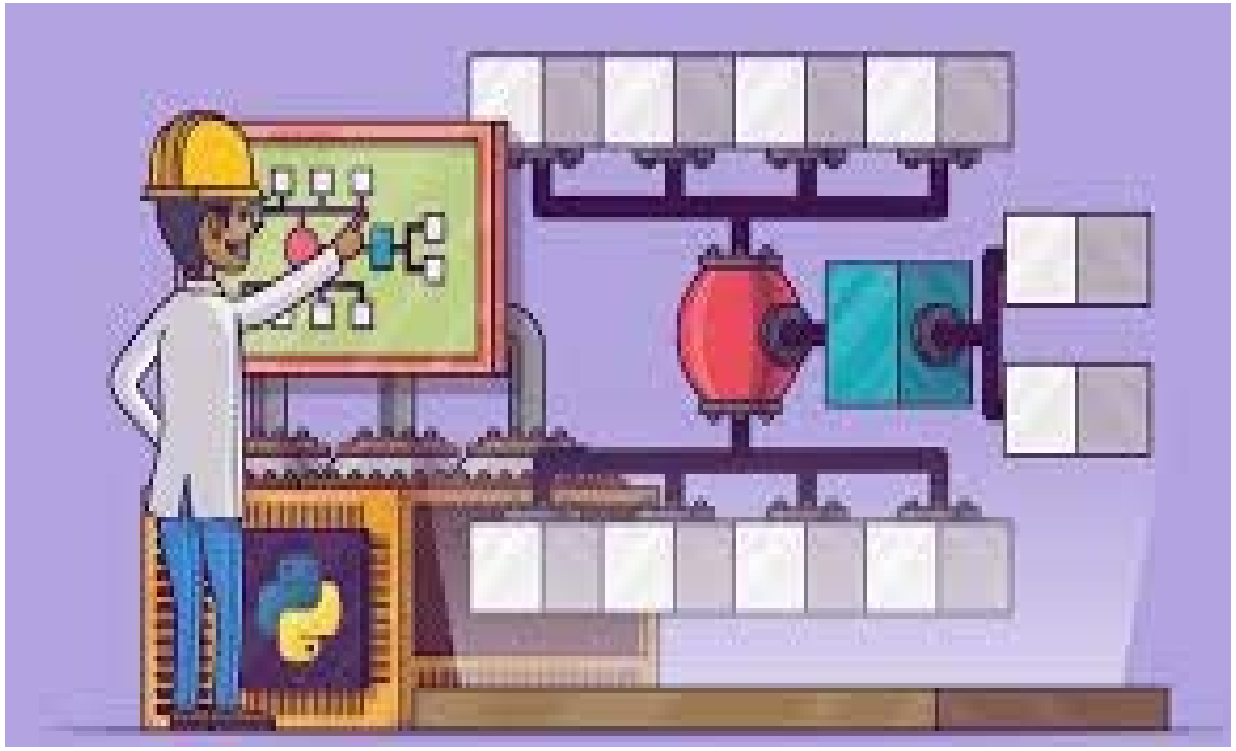
ساختمان داده ها



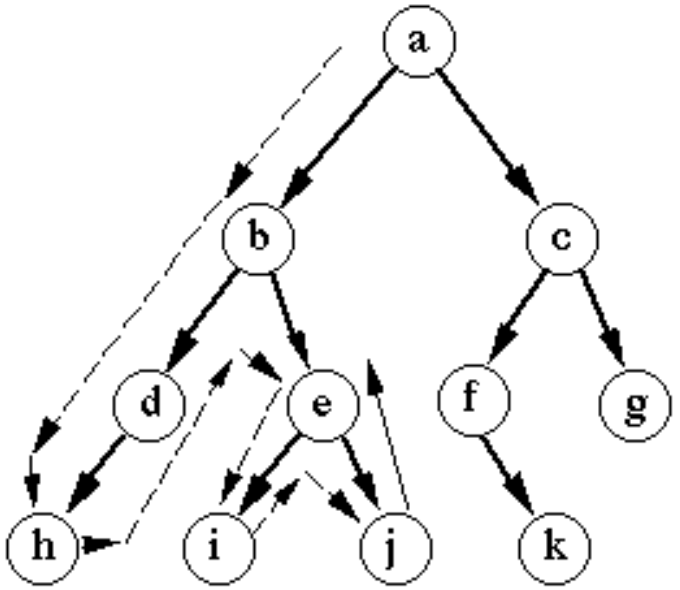
مدرس:
سمانه حسینی سمنانی

دانشگاه صنعتی اصفهان- دانشکده برق و کامپیوتر

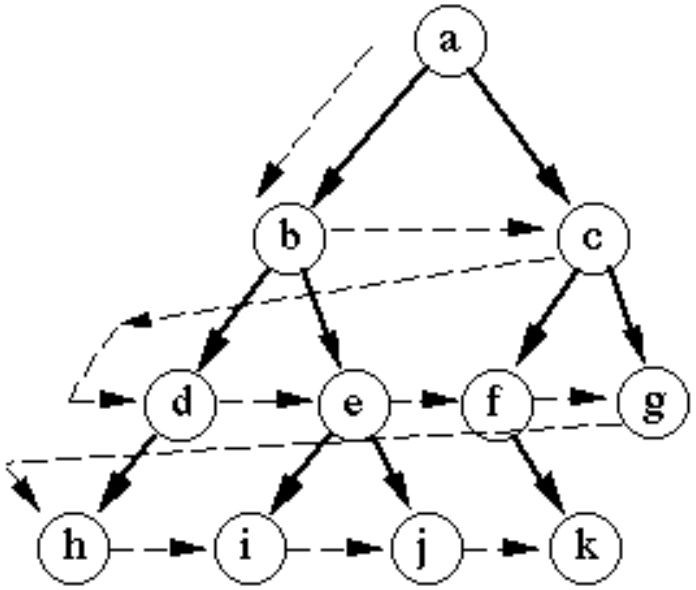
Elementary Graph Algorithms

- Graph representation
- graph-searching algorithm
  - breadth-first search
  - depth-first search
- minimum-spanning-tree
  - Kruskal
  - Prim
  - Sollin

Breadth-first search vs Depth-first search



Depth-first search



Breadth-first search

Depth-first search

- Search “deeper” in the graph whenever possible.
- Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it.
- Once all of v’s edges have been explored, the search “backtracks” to explore edges leaving the vertex from which v was discovered.
- This process continues until we have discovered all the vertices that are reachable from the original source vertex.
- If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source.
- The algorithm repeats this entire process until it has discovered every vertex.

Depth-first search

- As in breadth-first search, depth-first search colors vertices during the search to indicate their state.
- Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely.
- This technique guarantees that each vertex ends up in exactly one depth-first tree
- depth-first search also ***timestamps*** each vertex.
- Each vertex has two timestamps:
  - the first timestamp $v.d$ records when is first discovered (and grayed),
  - the second timestamp $v.f$ records when the search finishes examining 's adjacency list (and blackens).

Depth-first search

- For every vertex u:

$$u.d < u.f.$$

- Vertex u is WHITE before time $u.d$,
- GRAY between time $u.d$ and time $u.f$,
- BLACK thereafter.

- Timestamps are helpful in reasoning about the behavior of depth-first search.
- a global variable time.

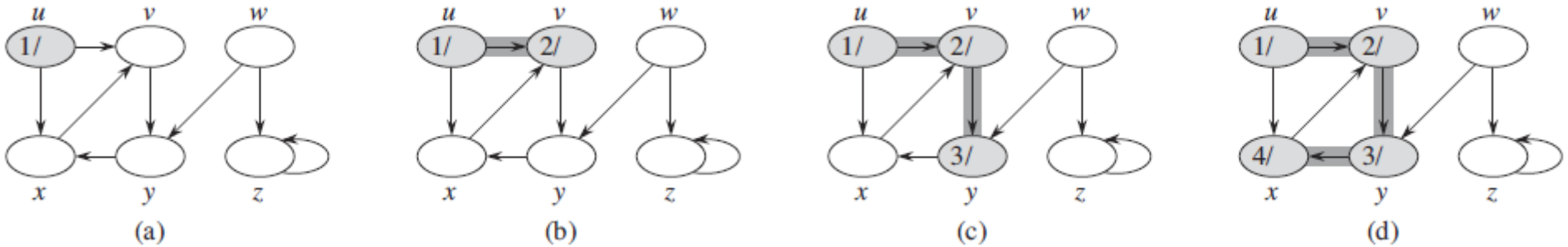
Depth-first search algorithm

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1  time = time + 1          // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]    // explore edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK          // blacken u; it is finished
9  time = time + 1
10 u.f = time
```

Depth-first search Example

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
```

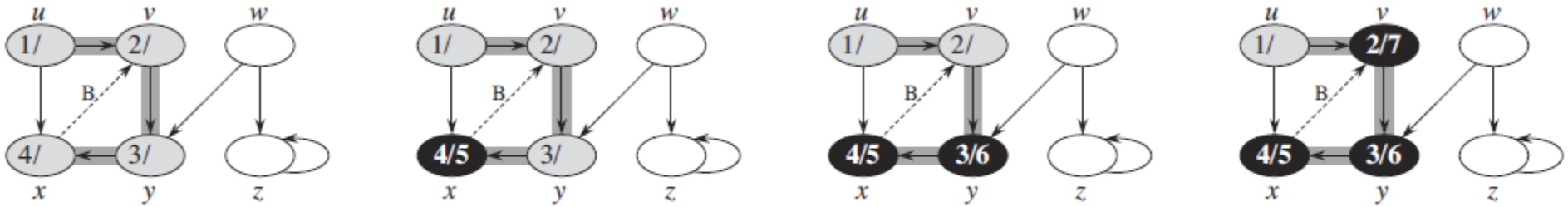


within vertices indicate discovery time/finishing times.

Depth-first search Example

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
```

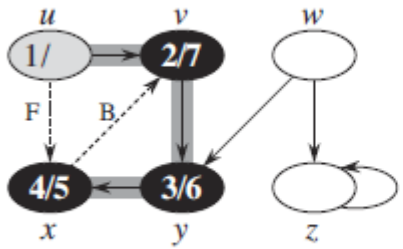
As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges.



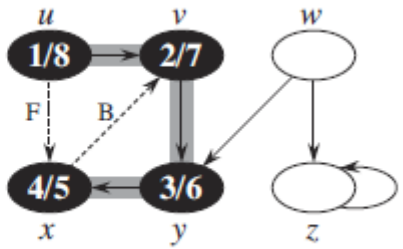
Depth-first search Example

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

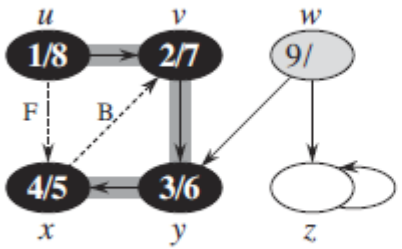
```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
```



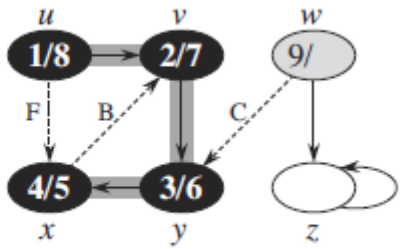
(i)



(j)



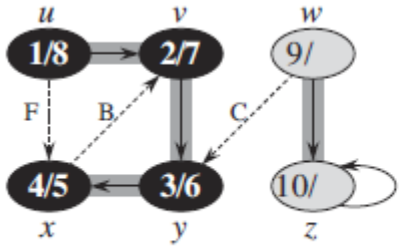
(k)



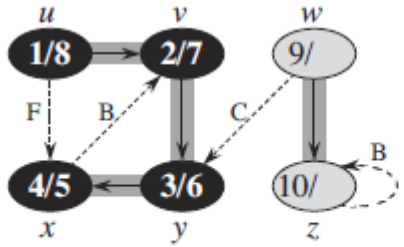
(l)

Depth-first search Example

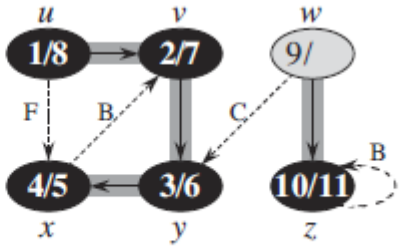
```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
```



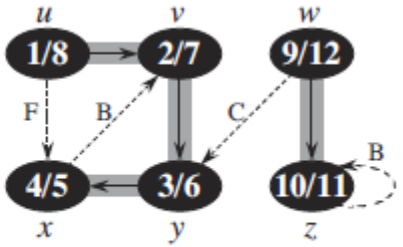
(m)



(n)



(o)



(p)

Depth-first search Example

- the results of depth-first search may depend upon the order in which line 5 of DFS examines the vertices
- and upon the order in which line 4 of DFS-VISIT visits the neighbors of a vertex.
- These different visitation orders tend not to cause problems in practice, as we can usually use any depth-first search result effectively, with essentially equivalent results.

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
```

DFS Analysis

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  u.color = BLACK
9  time = time + 1
```

- The loops on lines 1–3 and lines 5–7 of DFS take time $O(V)$
- The procedure DFS-VISIT is called exactly once for each vertex v $\in$ V , since the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray.
- The loop on lines 4–7 executes |Adj[v]| times.
- Since $\sum_{v \in V} |Adj[v]| = \Theta(E)$,

$\Theta(V + E)$.

سمانه حسینی سمنانی
هیات علمی دانشکده برق و کامپیوتر- دانشگاه صنعتی اصفهان

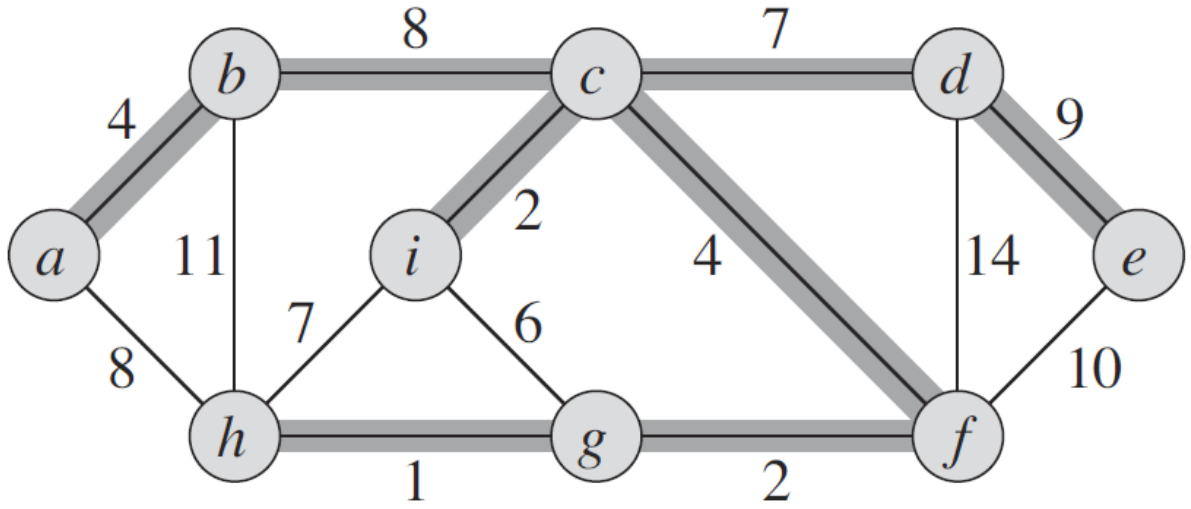
Minimum Spanning Trees

G = (V,E)

$w(u, v)$: the cost to connect u and v

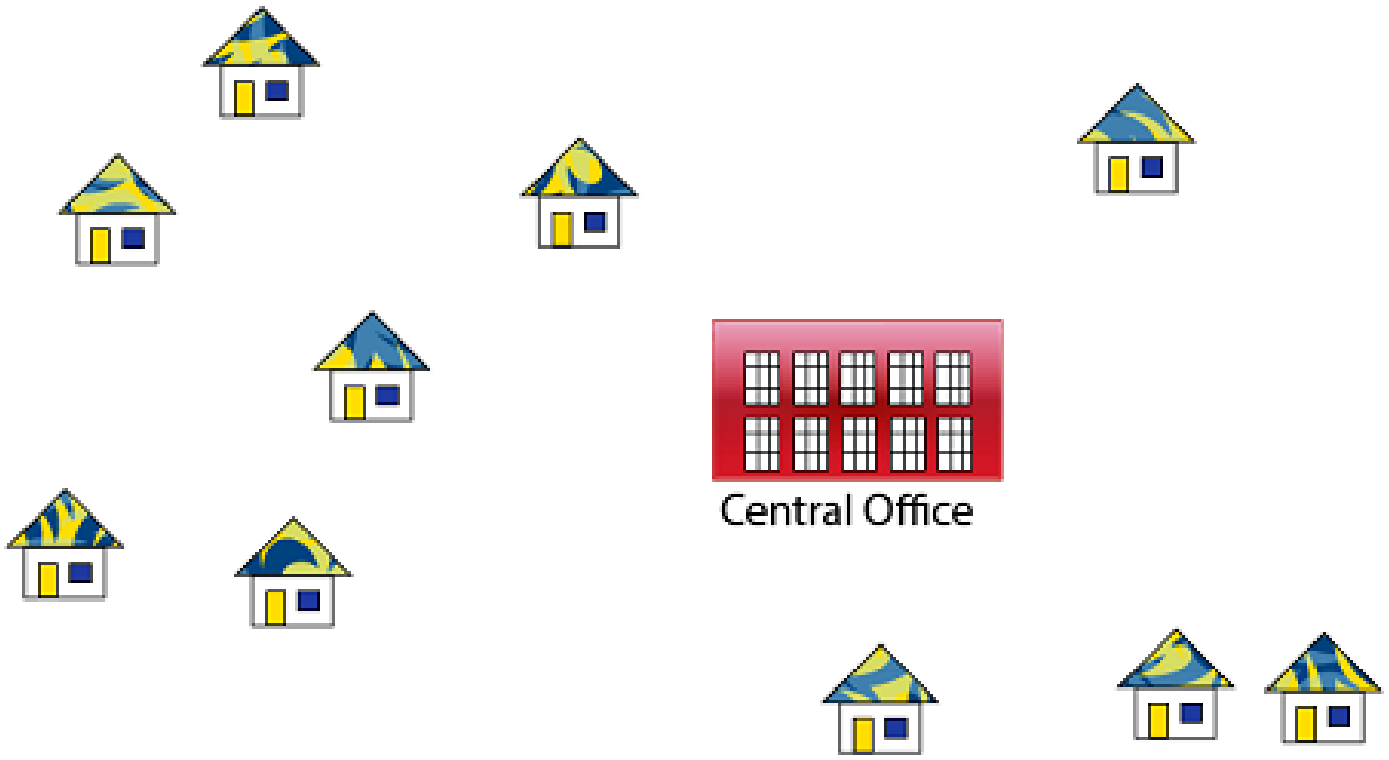
wish to find an acyclic subset $T \subseteq E$

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

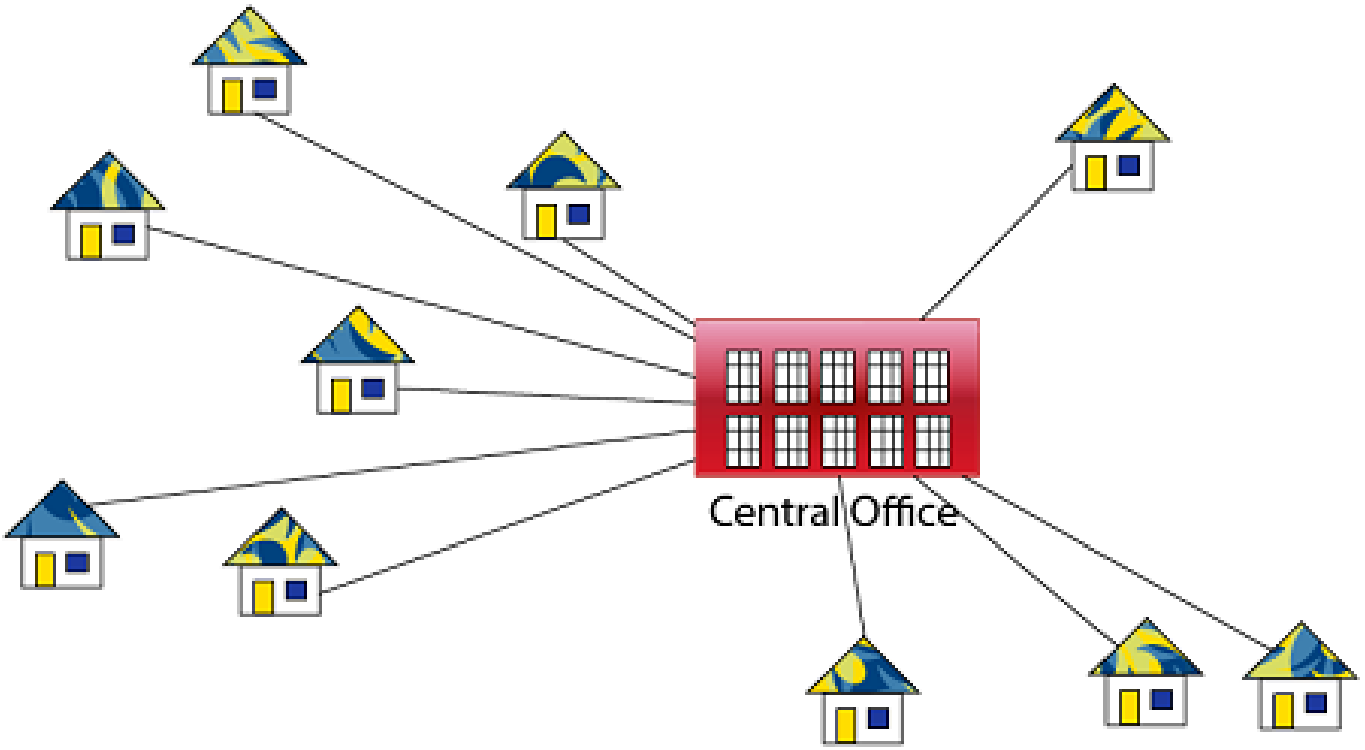


Since T is acyclic and connects all of the vertices, it must form a tree : ***spanning tree***

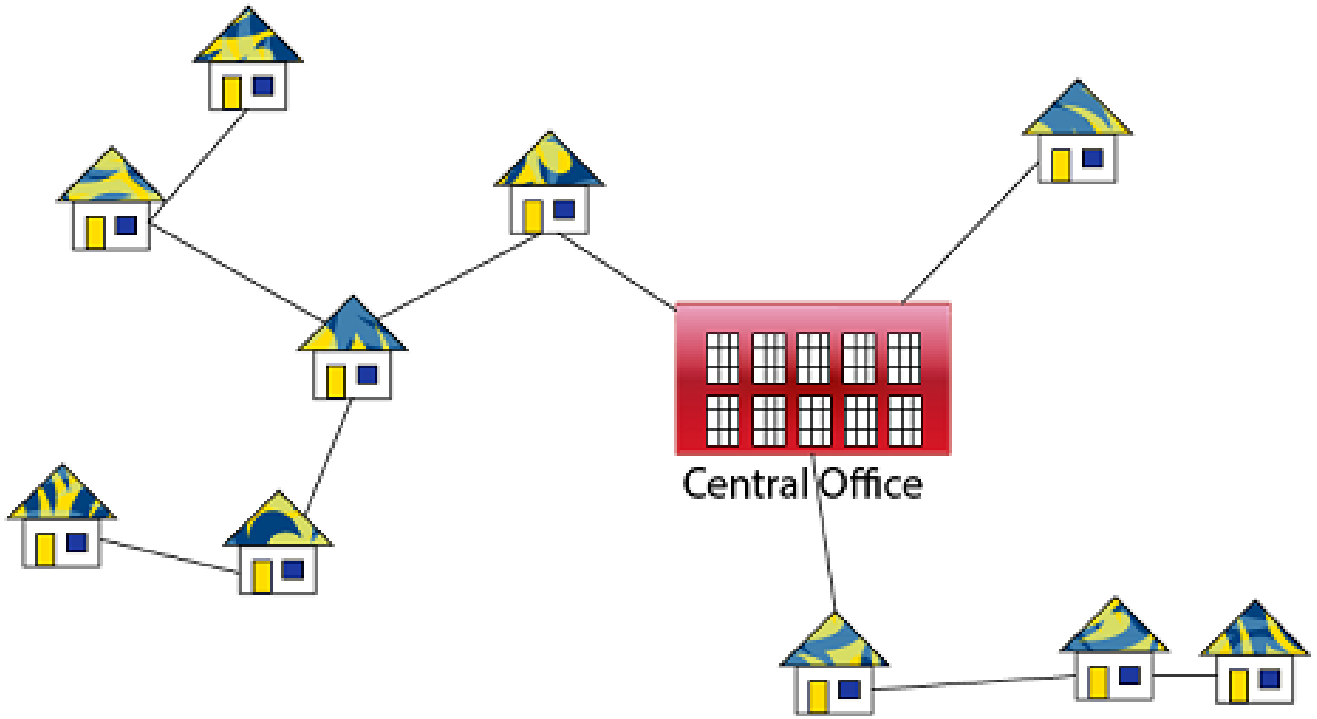
Ex. Problem laying Telephone Wire



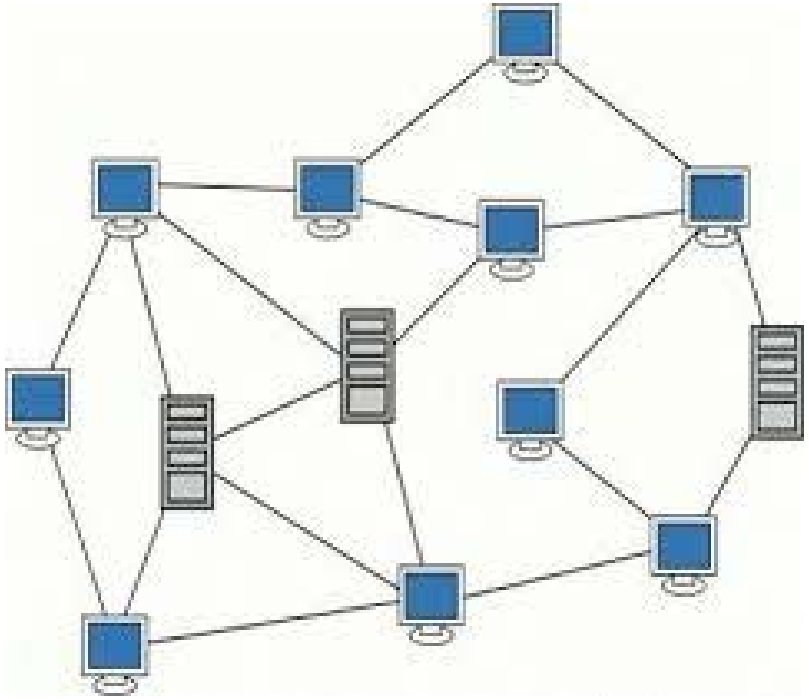
Naive approach: Expensive!



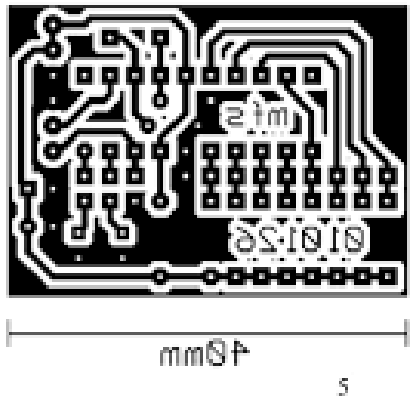
Minimize the total wire length



Minimum Spanning Trees



Electronic Circuits:



Growing a minimum spanning tree

$G = (V, E)$

Weight function: $w : E \to \mathbb{R}$.

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

The tricky part is, of course, finding a safe edge in line 3.

The algorithms of Kruskal, Prim and Sollin

```
GENERIC-MST(G, w)
1  A = ∅
2  while A does not form a spanning tree
3      find an edge (u, v) that is safe for A
4      A = A ∪ {(u, v)}
5  return A
```

each use a specific rule to determine a safe edge in line 3 of GENERIC-MST.

Kruskal's algorithm

- Kruskal's algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) of least weight.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

- The operation FIND-SET(u) returns a representative element from the set that contains u

Kruskal's algorithm

- Kruskal's algorithm finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u, v) of least weight.

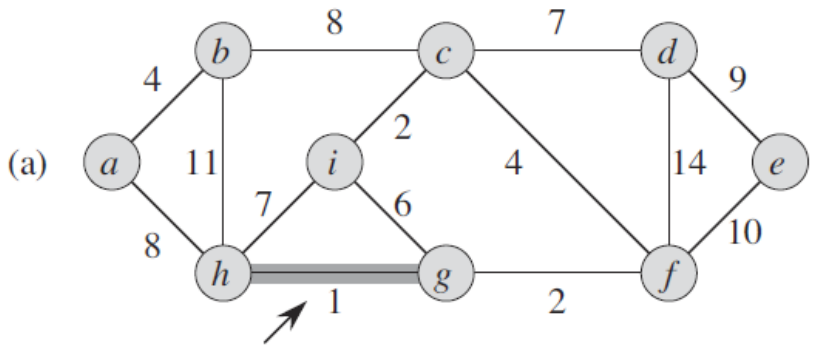
```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Thus, we can determine whether two vertices u and belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v)

- The operation FIND-SET(u) returns a representative element from the set that contains u

Kruskal's algorithm

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```



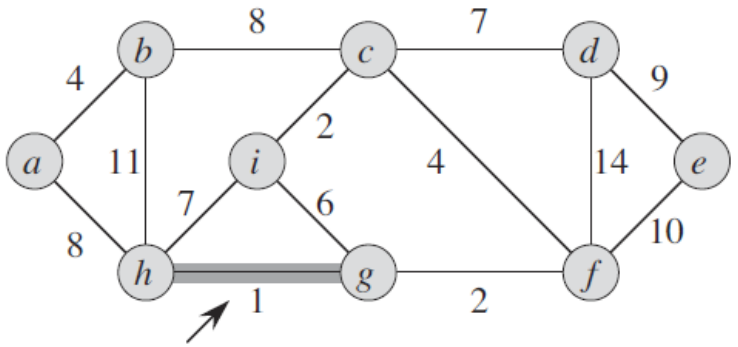
Lines 1–3 initialize the set A to the empty set and create |V| trees, one containing each vertex

for each edge (u, v) whether the endpoints u and v belong to the same tree.

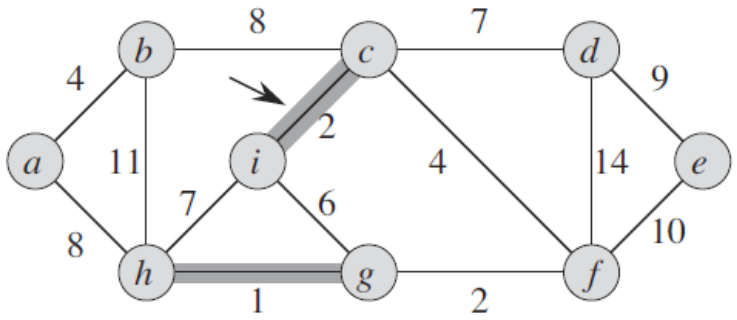
If they do, then the edge (u, v) cannot be added to the forest without creating a cycle, and the edge is discarded.

Kruskal’s algorithm

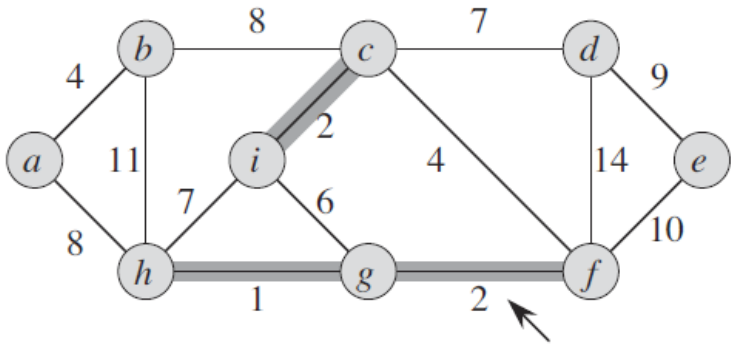
(a)



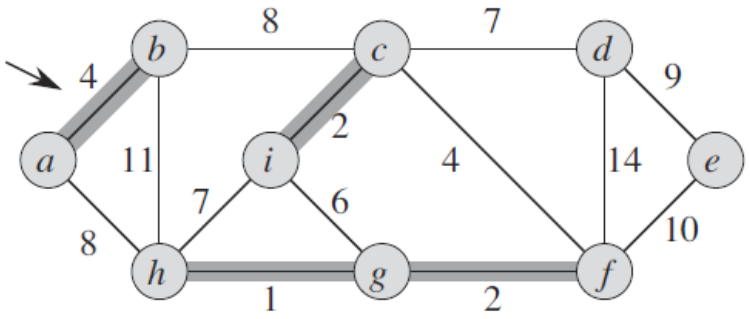
(b)



(c)

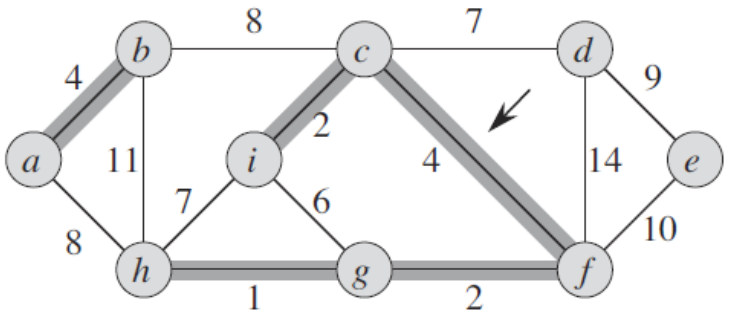


(d)

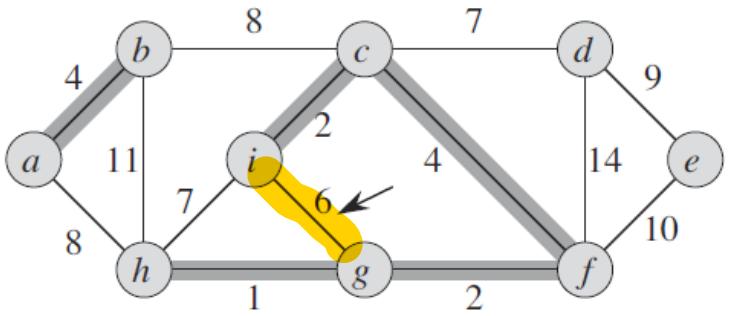


Kruskal’s algorithm

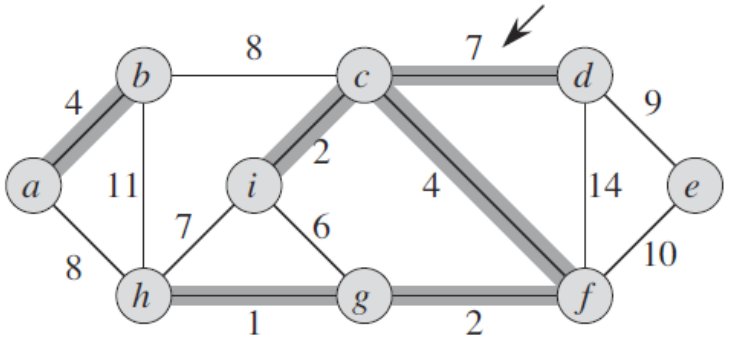
(e)



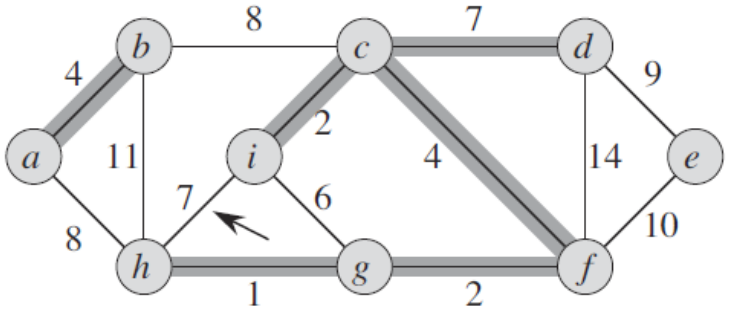
(f)



(g)

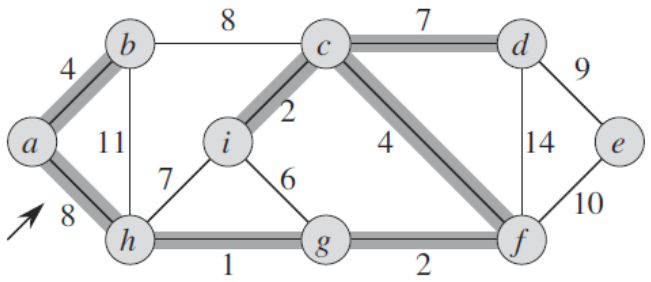


(h)

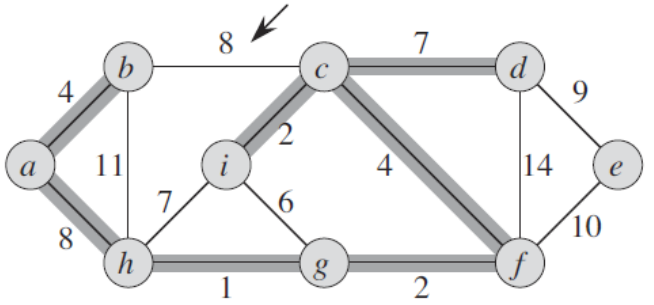


Kruskal’s algorithm

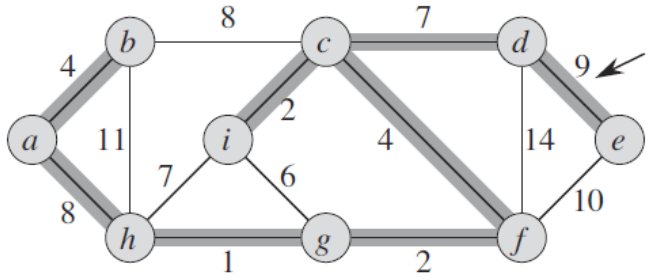
(i)



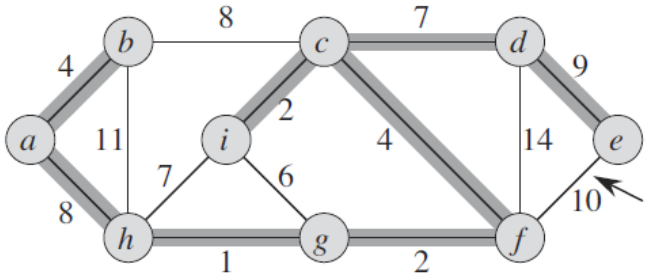
(j)



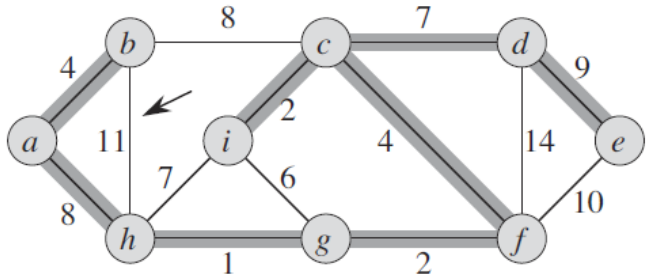
(k)



(l)



(m)



(n)