CSE5311 --- Lectures by Prof. Chris Ding

Graph Algorithms

Scribed by Huaisong Xu

Graph Theory Basics

Graph Representations

Graph Search (Traversal) Algorithms: BFS, DFS, Topological sort

Minimum Spanning Trees: Kruskal and Prim Algorithms

Single-Source Shortest Paths: Bellman-Ford, Dijkstra Algorithms

I Basic of Graph

Graph
A graph G is a triple consisting of a vertex set V(G), an edge set E(G), and a relation that associates with each edge two vertices (not necessarily distinct) called its endpoints.
Path
A path is a simple graph whose vertices can be ordered so that two vertices are adjacent if and only if they are consecutive in the list.
undirected
A graph in which each edge symbolizes an unordered, transitive relationship between two nodes. Such edges are rendered as plain lines or arcs.
directed, digraph
A graph in which each edge symbolizes an ordered, non-transitive relationship between two nodes. Such edges are rendered with an arrowhead at one end of a line or arc.
degree
The number of edges which connect a node.
In Degree:Number of edges pointing to a node.
Out Degree: Number of edges going out of a node.
unweighted
A graph in which all the relationships symbolized by edges are considered equivalent. Such edges are rendered as plain lines or arcs.
weighted (edge)
Weighted edges symbolize relationships between nodes which are considered to have some value, for instance, distance or lag time. Such edges are usually annotated by a number or letter placed beside the edge. If edges have *weights*, we can put the weights in the lists.

Weight: $W : E \rightarrow \mathbf{R}$

reachable/unreachable
A *vertex* v is reachable from another vertex u if there is a *path* of any length from u to v.

Note: Usually applied only to directed graphs, since any vertex in a connected, undirected graph is reachable from any other vertex.

II Graph Representation

Given graph $G = (V, E)$.

- May be either directed or undirected.
- Two common ways to represent for algorithms:

1. Adjacency lists.
2. Adjacency matrix.

When expressing the running time of an algorithm, it.s often in terms of both $|V|$ and $|E|$. In asymptotic notation.and *only* in asymptotic notation.we.ll drop the cardinality. Example: $O(V + E)$.

Adjacency lists

Array *Adj* of $|V|$ lists, one per vertex.

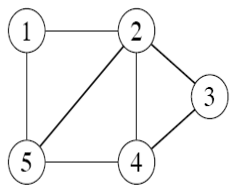
Vertex *u*.s list has all vertices v such that $(u, v) \in E$. (Works for both directed and undirected graphs.)

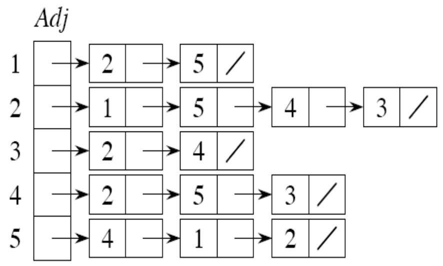
Adjacency matrix

$|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if}(i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Example: For an undirected graph:

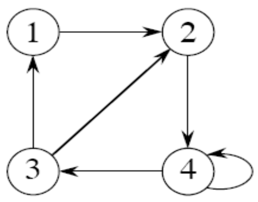


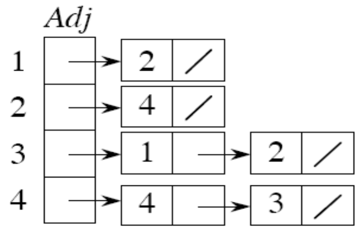


$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

.

Example: For a directed graph:





$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 1 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 \\ \hline \end{array}$$

Counting neighbors

A. Counting the number of common neighbors between i,j = # of 2-hop paths between i,j:

$$(A^2)_{ij} = \sum_{K=1}^{n} A_{ik} \times A_{kj}$$

B. Counting the # of 3-hop paths between i,j:

$$(A^3)_{ij} = \sum_{kl} A_{ik} \times A_{kl} \times A_{lj}$$

III Graph Search (Traversal) Algorithms

There are two types of graph search algorithms: Bread First Search and Depth first Search.

1. Bread First Search

It is common used, because of its simple intuitive characteristic, finding the node's neighbor, then neighbor's neighbor.

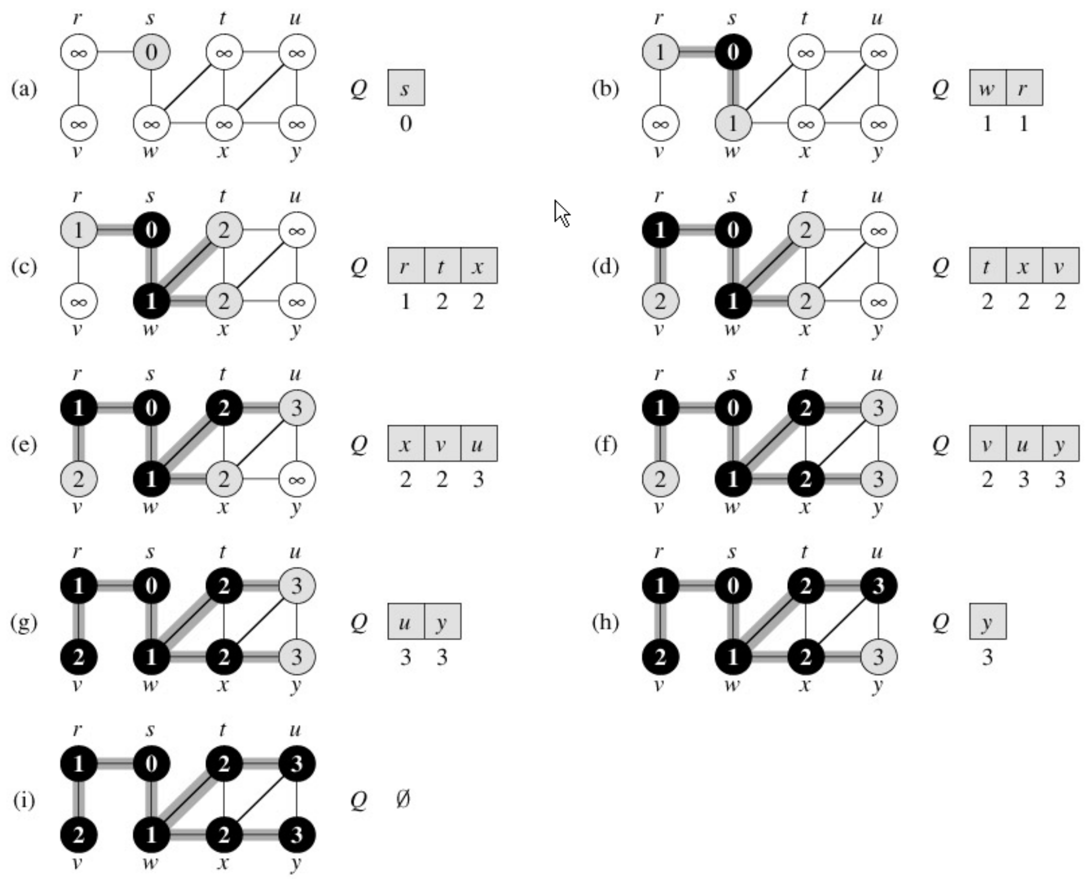
```
BFS(V, E, s)
for each u $\in$ V $-\{s\}$
    do color[u]$\leftarrow$white ; d[u]=$\infty$; $\pi$[u]=NIL;
end for
Color[s]=Gray;d[s] $\leftarrow$ 0; $\pi$[s]=NIL
Q $\leftarrow \varnothing$;
ENQUEUE (Q, s)
while Q _= $\varnothing$
    do u $\leftarrow$ DEQUEUE(Q)
        for each v $\in$ Adj[u]
            d[v] $\leftarrow$ d[u] + 1;
            color[v]=gray;
            $\pi$[v]=u;
            ENQUEUE (Q, v);
        end for
        color[u]=black
end while
```

Color:
white----Not be touched
Gray----Touched
Black----Touched and neighbor
touched exauhxieve

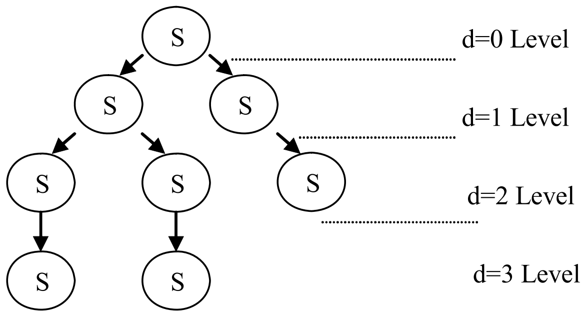
$\delta(s,v) \le d[v]$

$\delta(s,u) \le \alpha(s,v) + 1$

The total **running time** of BFS is $O(V+E)$


(a) Q s 0
(b) Q w r 1 1
(c) Q r t x 1 2 2
(d) Q t x v 2 2 2
(e) Q x v u 2 2 3
(f) Q v u y 2 3 3
(g) Q u y 3 3
(h) Q y 3
(i) Q Ø
r s t u v w x y

BFS Tree



Shortest-path distance δ(*s*, *v*) from *s* to *v* as the minimum number of edges in any path from vertex *s* to vertex *v*; if there is no path from *s* to *v*, then δ(*s*, *v*) = ∞. A path of length δ(*s*, *v*) from *s* to *v* is said to be a *shortest path*[1] from *s* to *v*.

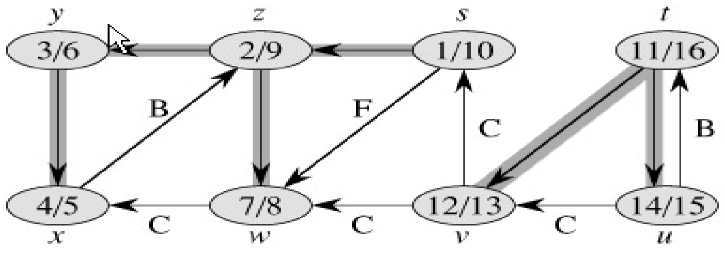
2. Depth First Search

The strategy followed by depth-first search is, to search "deeper" in the graph whenever possible.

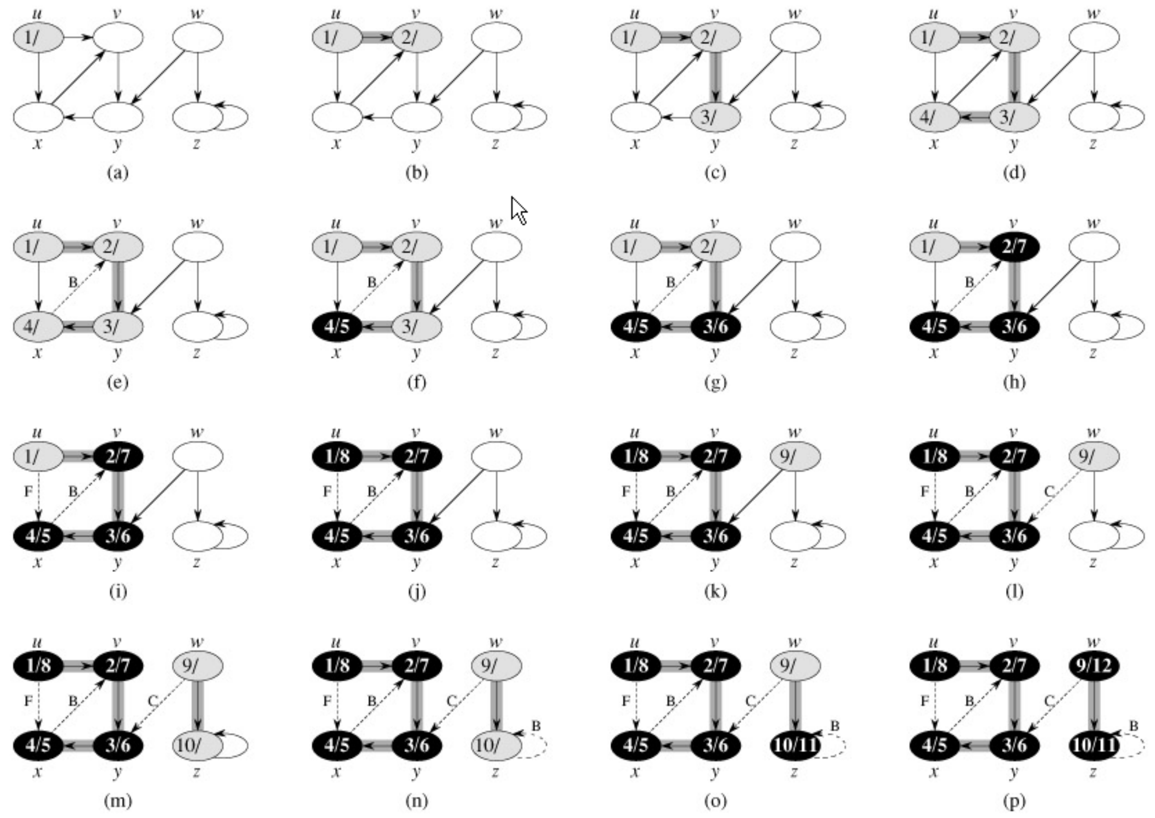
$\pi[u]$ is array containing predesessors

```
DFS(G)
1   for each vertex u ∈ V[G]
2           do color[u] ← WHITE
3                π[u] ← NIL
4   time ← 0
5   for each vertex u ∈ V[G]
6           do if color[u] = WHITE
7                   then DFS-VISIT(u)
DFS-VISIT(u)
1   color[u] ← GRAY          ▹White vertex u has just been discovered.
2   time ← time +1
3   d[u] time
4   for each v ∈ Adj[u]    ▹Explore edge(u, v).
5           do if color[v] = WHITE
6                   then π[v] ← u
7                                 DFS-VISIT(v)
8   color[u] BLACK           ▹ Blacken u; it is finished.
9   f [u] ▹ time ← time +1
```

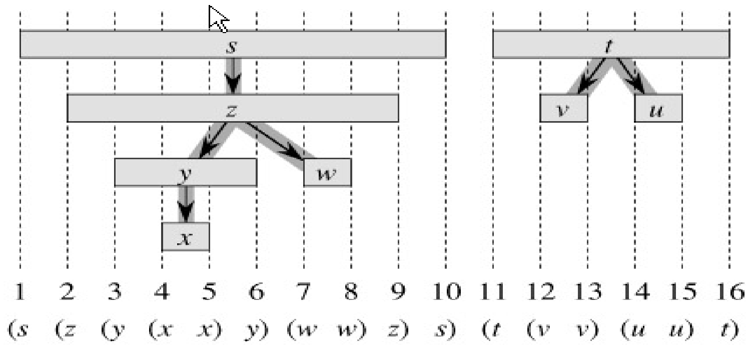
The final result is



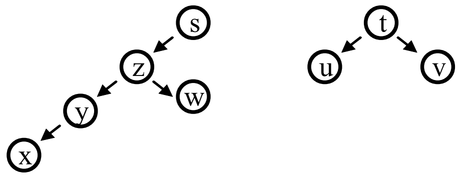
The procedure



Parenthesis Structure: A important property of DFS is discovered and showed as below.



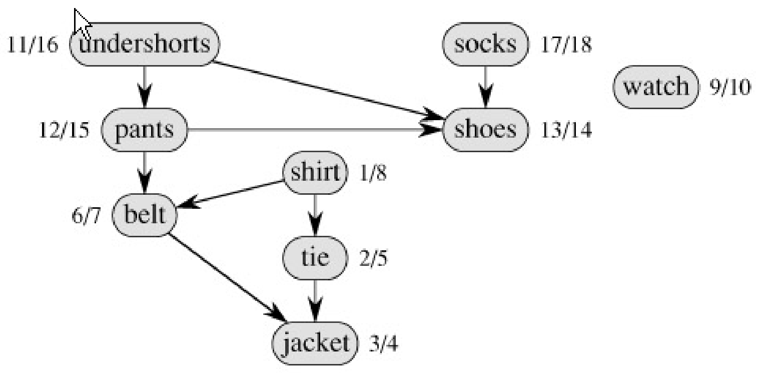
DFS Tree: It maybe generate multiple trees, forest.

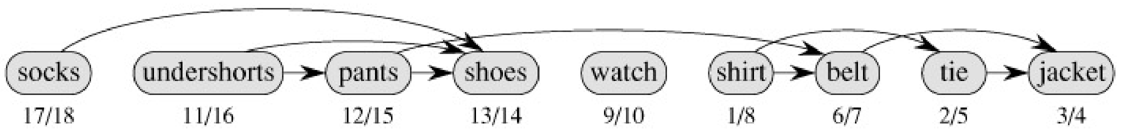


The running time of DFS is therefore $\Theta(V + E)$.

3. Topological Sort

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





```
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
```

We can perform a topological sort in time $\Theta(V + E)$, since depth-first search takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

IV Single-Source Shortest Paths

Single-source shortest-paths problem: given a weighted (unweighted graph could be treated as a weight graph that weight of every edge is 1), directed graph $G = (V, E)$, we want to find a shortest path from a given *source* vertex $s \in V$ to each vertex $v \in V$.

Both Bellman-Ford algorithm and Dijkstra algorithm will use **Relaxation algorithm**. It is defined as below:

IF $d(s,v) > d(s,u) + w(u,v)$

Then $d(s,v) = d(s,u) + w(u,v)$;

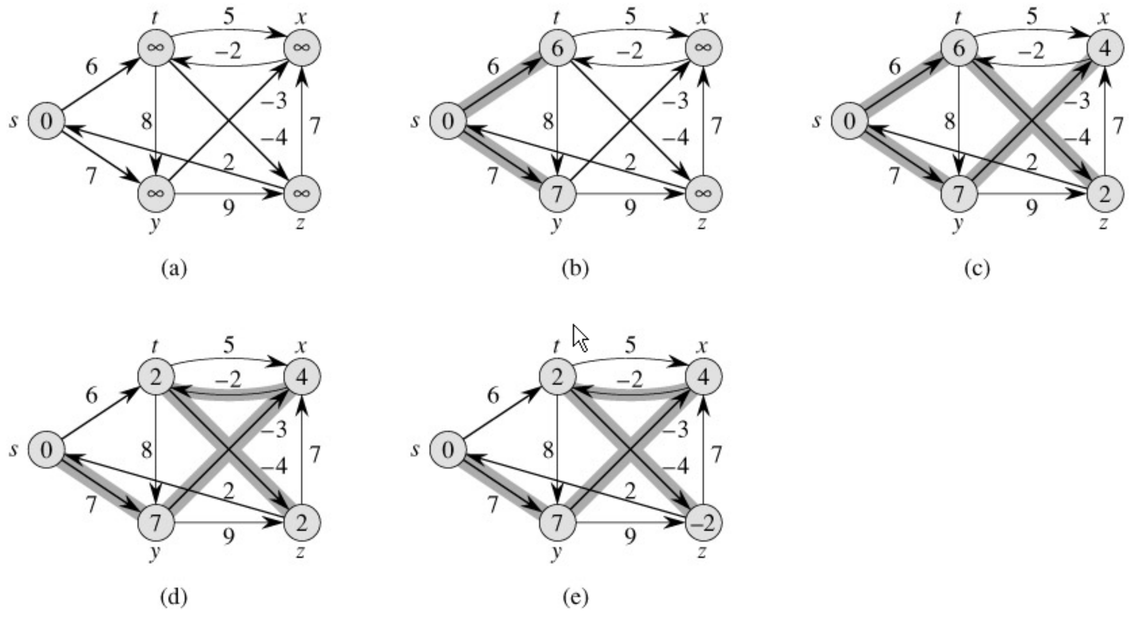
Also, we would know the **Triangle inequality**

- For any edge $(u, v) \in E$, we have $\delta(s, v) \le \delta(s, u) + w(u, v)$

1. The Bellman-Ford algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to |V[G]| - 1
```

```
3         do for each edge $(u, v) \in E[G]$
4                do RELAX$(u, v, w)$
5   for each edge $(u, v) \in E[G]$
6         do if $d[v] > d[u] + w(u, v)$
7               then return FALSE
8   return TRUE
```



The Bellman-Ford algorithm runs in time $O(V E)$, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V|$ - 1 passes over the edges in lines 2-4 takes $\Theta(E)$ time, and the for loop of lines 5-7 takes $O(E)$ time.

2. Dijkstra's algorithm

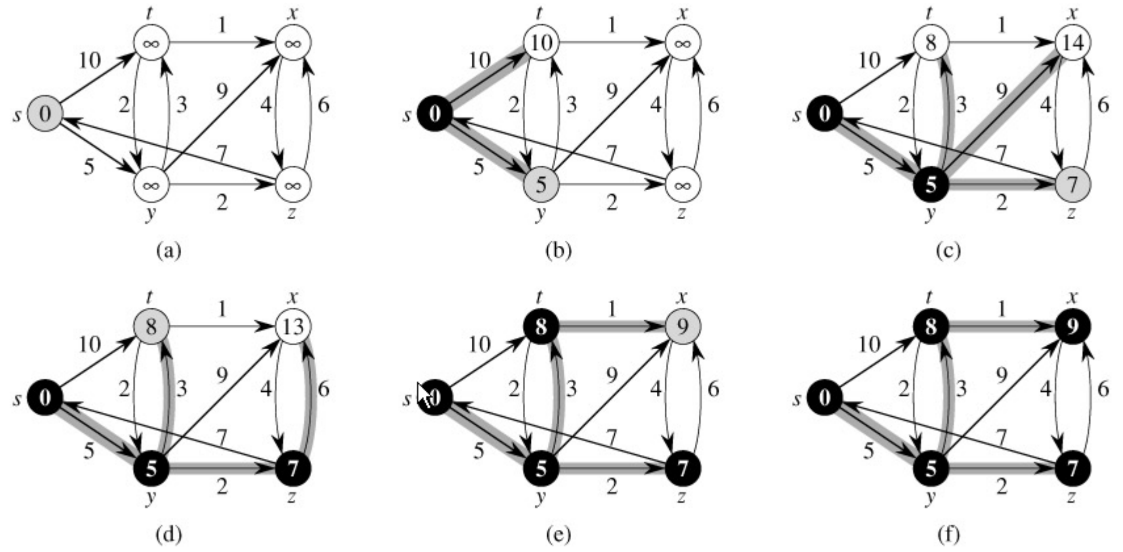
Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. The running time of Dijkstra's algorithm is lower than that of the Bellman-Ford algorithm.

- Maintains a set S of vertices whose final shortest-path weights from the source s have already been determined.
- Repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.

In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

```
DIJKSTRA$(G, w, s)$
1   INITIALIZE-SINGLE-SOURCE$(G, s)$
2   $S \leftarrow \emptyset$
3   $Q \leftarrow V[G]$
4   while $Q \neq \emptyset$
```

```
5        do $u \leftarrow$ EXTRACT-MIN($Q$)
6            $S \leftarrow S \cup \{u\}$
7            for each vertex $v \in Adj[u]$
8                do RELAX($u$, $v$, $w$)
```



The running time of Dijkstra's algorithm is $O(V^2+E) = O(V^2)$.

V Minimum Spanning Trees

The purpose of MST problem is to find the shortest path, that all nodes on the path.

Hamilton Path: Go through all nodes, but never come back, one node one time. If over one time ,then become Hamiltonian Cycle.

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

We will discuss two algorithms, Kruskal's algorithm and Prim's algorithm.

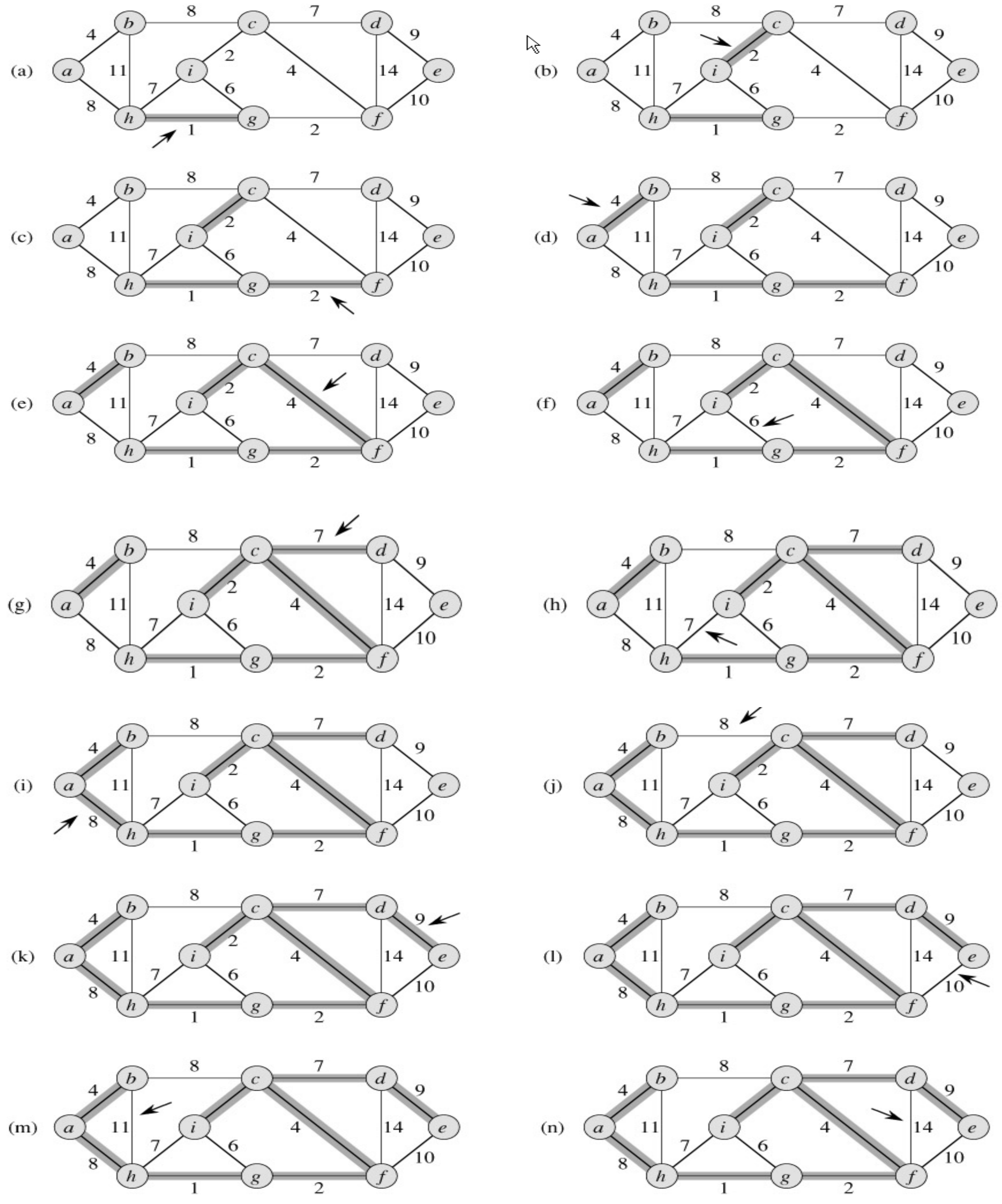
In Kruskal's algorithm, the set *A* is a forest. The safe edge added to *A* is always a least-weight edge in the graph that connects two distinct components.

In Prim's algorithm, the set *A* forms a single tree. The safe edge added to *A* is always a least-weight edge connecting the tree to a vertex not in the tree.

1. Kruskal's algorithm

```
MST-KRUSKAL($G$, $w$)
1  $A \leftarrow \emptyset$
2  for each vertex $v \in V[G]$
3        do MAKE-SET($v$)
```

```
4   sort the edges of $E$ into nondecreasing order by weight $w$
5   for each edge $(u, v) \in E$, taken in nondecreasing order by weight
6           if FIND-SET($u$) $\neq$ FIND-SET($v$)
7               then $A \leftarrow A \cup \{(u, v)\}$
8                   UNION($u$, $v$)
9   return $A$
```

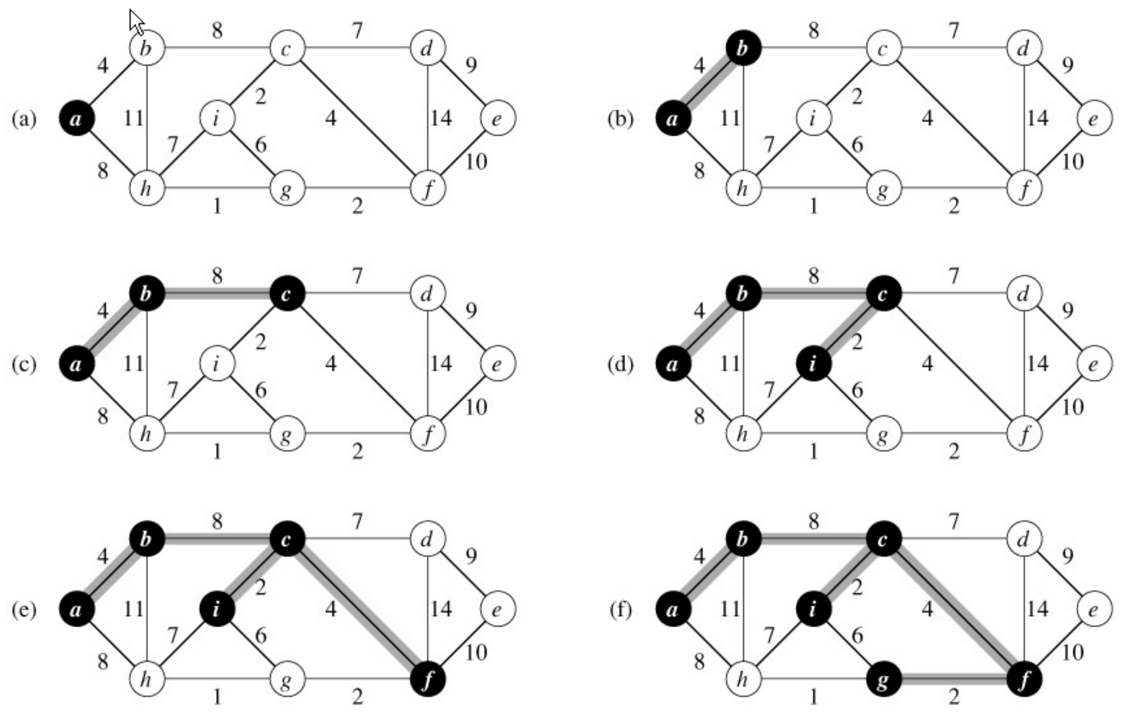


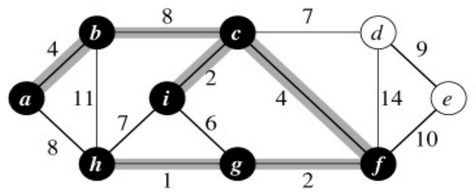
All edge weight: 1 2 2 4 4 ~~6~~ ~~7~~ 7 8 ~~8~~ 9 ~~10~~ ~~11~~ ~~11~~

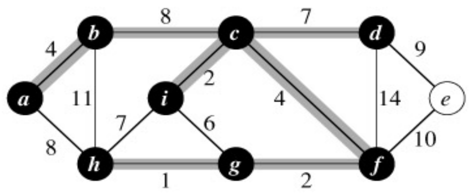
subtree_ID= Is_in_a_subtree: Is the edge between two nodes, which are already in Subtree, if in, don't add them to subtree, otherwise add them to subtree. Obviously it is not unique.

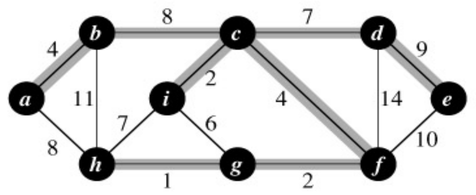
2. Prim's algorithm

```
MST-PRIM($G$, $w$, $r$)
for each $u \in V[G]$
        do $key[u] \leftarrow \infty$
            $\pi[u] \leftarrow$ NIL
$key[r] \leftarrow 0$
 $Q \leftarrow V[G]$
 while $Q \neq \emptyset$
        do $u \leftarrow$ EXTRACT-MIN($Q$)
            for each $v \in Adj[u]$
                 do if $v \in Q$ and $w(u, v) < key[v]$
                        then $\pi[v] \leftarrow u$
                               $key[v] \leftarrow w(u, v)$
```



(g)


(h)


(i)


The running time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asymptotically the same as for the implementation of Kruskal's algorithm. However, the asymptotic running time of Prim's algorithm can be improved, however, by using Fibonacci heaps. We can perform an EXTRACT-MIN operation in $O(\lg V)$ amortized time and a DECREASE-KEY operation (to implement line 11) in $O(1)$ amortized time. Therefore, if we use a Fibonacci heap to implement the min-priority queue Q, the running time of Prim's algorithm improves to $O(E + V \lg V)$.