# CS302: Design and Analysis of Algorithms

## DFS traversals

# DFS

- Depth-first search is a systematic way to find all the vertices reachable from a source vertex, s.
- DFS will process the vertices first deep and then wide. After processing a vertex it recursively processes all of its descendants
- Like BFS, to keep track of progress depth-first-search colors each vertex.
- Each vertex of the graph is in one of three states:
  1. Undiscovered;
  2. Discovered but not finished (not done exploring from it); and
  3. Finished (have found everything reachable from it) i.e. fully explored.

- The state of a vertex, *u*, is stored in a color variable as follows:
  1. color[*u*] = White - for the "undiscovered" state,
  2. color[*u*] = Gray - for the "discovered but not finished" state, and
  3. color[*u*] = Black - for the "finished" state.
- Depth-first search uses π[*v*] to record the parent of vertex *v*. We have π[*v*] = NIL if and only if vertex *v* is the root of a depth-first tree.
- DFS time-stamps each vertex when its color is changed.

  1. When vertex *v* is changed from white to gray the time is recorded in d[*v*].
  2. When vertex *v* is changed from gray to black the time is recorded in f[*v*].

# Algorithm Depth-First Search

- The DFS forms a depth-first forest comprised of more than one depth-first trees.
- Each tree is made of edges ($u$, $v$) such that $u$ is gray and $v$ is white when edge ($u$, $v$) is explored.
- The following pseudocode for DFS uses a global timestamp time.

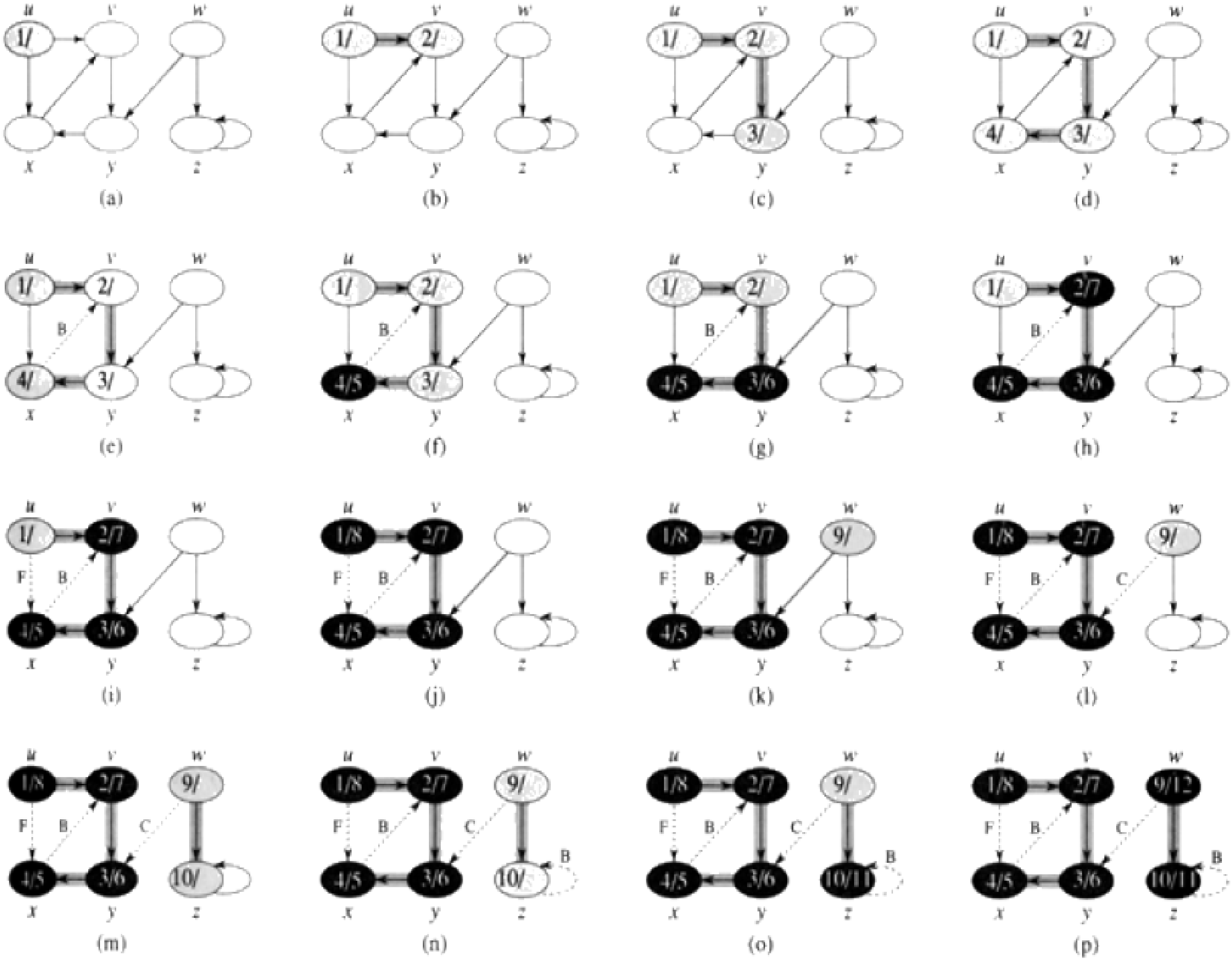
**DFS (G)**

```
for each vertex u in V[G]
    do color[u] ← WHITE
        π[u] ← NIL
time ← 0
for each vertex u in V[G]
    do if color[u] ← WHITE
        then DFS-Visit(u)                ▷ build a new DFS-tree from u
```

**DFS-Visit(u)**

```
color[u] ← GRAY ▷ discover u
time ← time + 1
d[u] ← time
for each vertex v adjacent to u ▷ explore (u, v)
  do if color[v] ← WHITE
        then π[v] ← u
         DFS-Visit(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time ▷ we are done with u
```

- **Example**:
- In the following figure, the solid edge represents discovery or tree edge and the dashed edge shows the back edge. Furthermore, each vertex has two time stamps: the first time-stamp records when vertex is first discovered and second time-stamp records when the search finishes examining adjacency list of vertex.


u v w x y z
1/ 2/ 3/ 4/
4/5 3/6 2/7 1/8 9/ 10/ 10/11 9/12
B F C
(a) (b) (c) (d)
(e) (f) (g) (h)
(i) (j) (k) (l)
(m) (n) (o) (p)

# Analysis

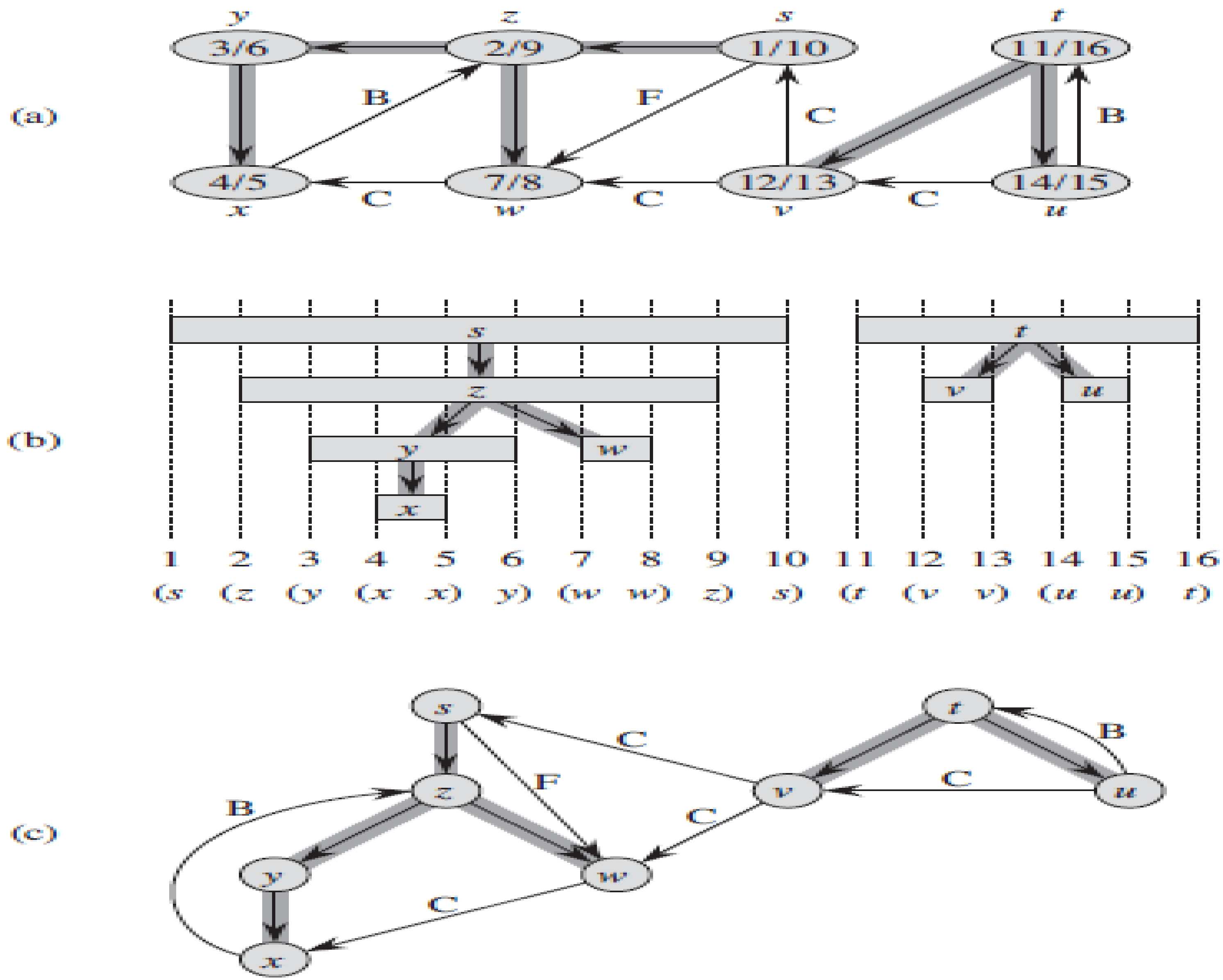
- The for-loop in DFS-Visit is executed a total of $|E|$ times for a directed graph or $2|E|$ times for an undirected graph since each edge is explored once.
- Initialization takes $\Theta(|V|)$ time. Therefore, the running time of DFS is $\Theta(V + E)$.

# Classification of edges

- Consider a directed graph G = (V, E). After a DFS of graph G we can put each edge into one of four classes:
  1. A **tree edge** is an edge in a DFS-tree.
  2. A **back edge** connects a vertex to an ancestor in a DFS-tree. Note that a self-loop is a back edge.
  3. A **forward edge** is a non-tree edge that connects a vertex to a descendent in a DFS-tree.
  4. A **cross edge** is any other edge in graph G. It connects vertices in two different DFS-tree or two vertices in the same DFS-tree neither of which is the ancestor of the other

# Parenthesis Theorem

- For all $u$, $v$, exactly one of the following holds:
  1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither of $u$ and $v$ is a descendant of the other.
  2. $d[u] < d[v] < f[v] < f[u]$ and $v$ is a descendant of $u$.
  3. $d[v] < d[u] < f[u] < f[v]$ and $u$ is a descendant of $v$.




a)DFS search   b)Intervals for the discovery time and finishing time of each vertex correpond to the parenthesization shown  c)classification of edges

- **White-path Theorem** Vertex *v* is a descendant of *u* if and only if at time d[*u*], there is a path *u* to *v* consisting of only white vertices. (Except for *u*, which was just colored gray.)
-

# Applications of DFS or Depth First Search

- If we perform DFS on unweighted graph, then it will create minimum spanning tree for all pair shortest path tree
- We can detect cycles in a graph using DFS. If we get one back-edge during BFS, then there must be one cycle.
- Using DFS we can find path between two given vertices u and v.
- We can perform topological sorting is used to scheduling jobs from given dependencies among jobs. Topological sorting can be done using DFS algorithm.
- Using DFS, we can find strongly connected components of a graph. If there is a path from each vertex to every other vertex, that is strongly connected.