Graph Traversal

- Just as there were techniques for traversing trees, there are techniques for traversing a graph
- Start at one vertex; do something there; go to next vertex; do something there; until we have visited all vertices
- Be careful to avoid re-visiting nodes
  - We have to mark a node as having been visited
  - When drawing this, we would "color" each node

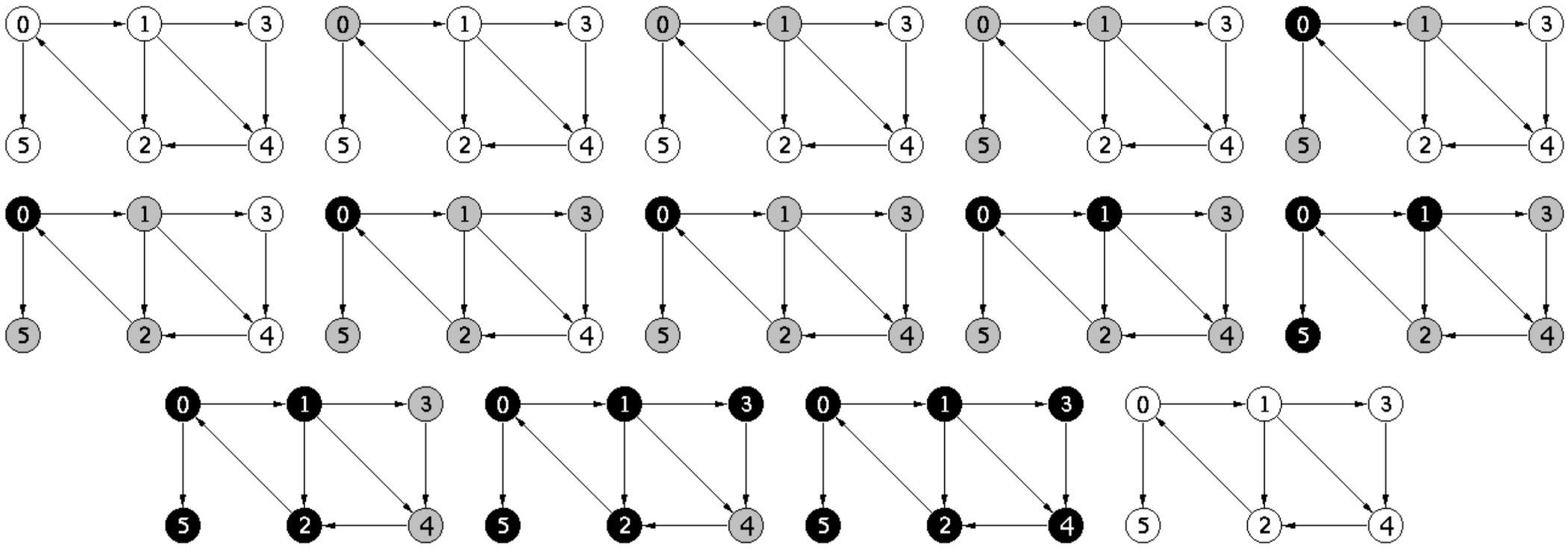
Breadth-first search

- Goal: always visit adjacent nodes first
- Color scheme:
  - white: a newly discovered node
  - gray: a discovered node which has not been completely processed
  - black: a completely processed node

BFS(G, s)
Input: G, a graph; s, the source vertex from which to begin BFS
Purpose: visits all vertices of G reachable from s

```
    for each vertex v in Vertices(G)
        color[v] = white
    color[s] = gray
    enqueue s into Q
    while Q is not empty
        v = Q.peek()
        for each u adjacent to v
            if color[u] = white     // Found a new node
                color[u] = gray     // Mark it as discovered
                enqueue u into Q    // Enqueue for later exploration
        dequeue from Q
        color[v] = black            // Mark v as fully explored
```

- Example (note: when deciding which vertex to next visit, choose the lower numbered vertex):



- If we were to print nodes in breadth-first order, starting from 0, we would get: 0, 1, 5, 3, 4, 2

Depth First Search

- Start at one vertex: begin exploring it by going to one neighbor
  - begin exploring a single neighbor of this new vertex
    - begin exploring a single neighbor from this node
- Backtrack when necessary
- Uses recursion

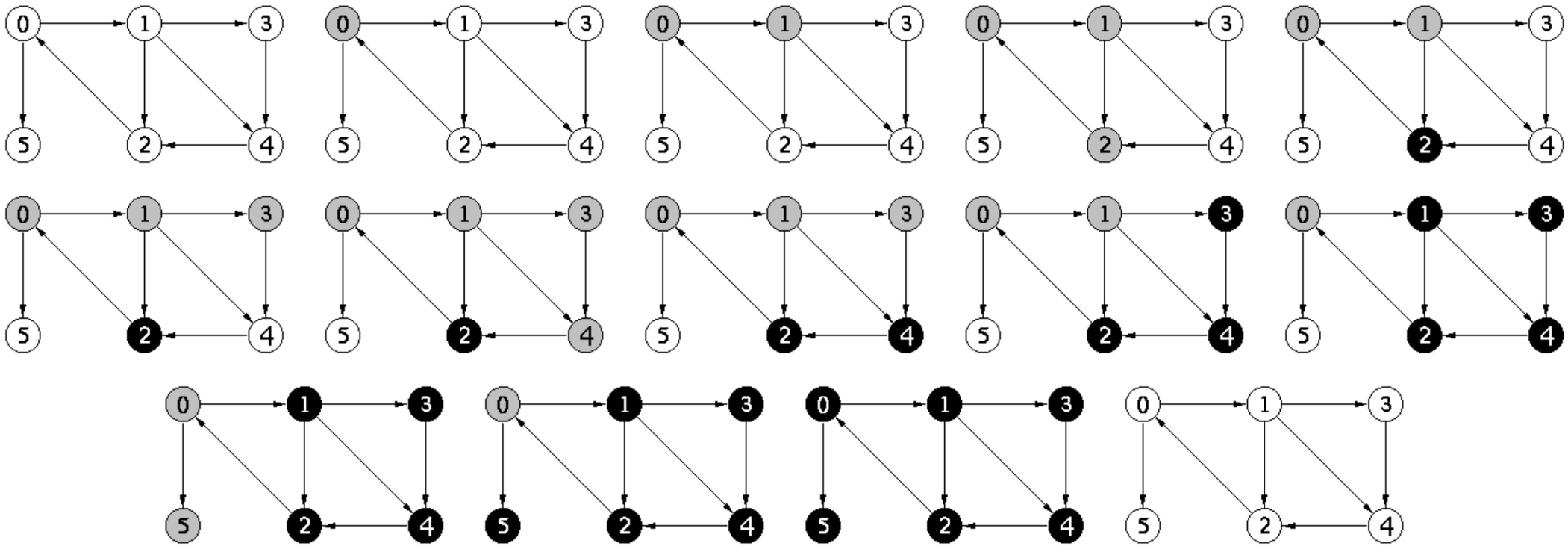
DFS(G, s)
Input: A graph, G; a source vertex s
Purpose: visits all vertices of G reachable from s

```
    for each vertex v in Vertices(G)
        color[v] = white
    DFS_Visit(s)
```

DFS_Visit(v)
Input: the node currently being visited, v
Postcondition: All nodes reachable from v will have been visited

```
    color[v] = gray             // Mark v as discovered
    for each u adjacent to v
        if color[u] == white
            DFS_Visit(u)        // Found a new node; explore it
    color[v] = black            // Mark v as fully explored
```

- Example:



Topological Sort

- If we have a DAG, we can "topologically sort our graph"
- Produces an "ordered graph"
  - If G contains an edge (u, v), then u appears before v when ordered
- How: perform a slightly modified DFS
  - Need new data structures for "discovery time" (d), "finishing time" (f)
  - Need to visit all nodes; not starting from a specific source
- Find finishing time from modified DFS; as each vertex is finished, insert node at front of linked list

DFS(G, s)
Input: A Graph, G; a source node s

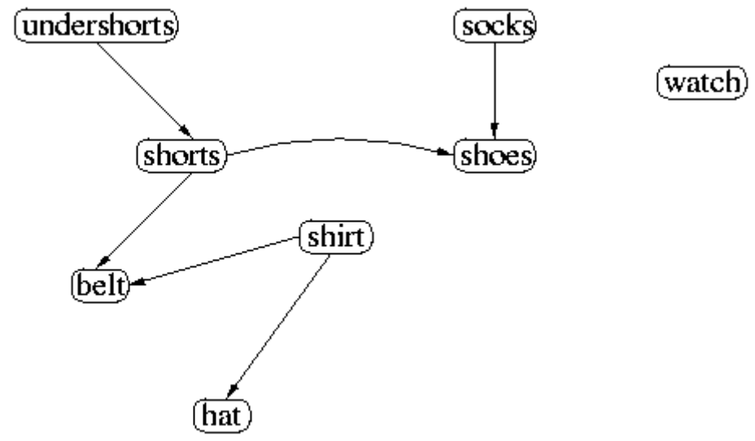
Purpose: visits all nodes of G reachable from s

```
    time = 0
    for each vertex v in Vertices(G)
        color[v] = white
    for each vertex v in Vertices(G)
        if color[v] == white
            DFS_Visit(v)
```

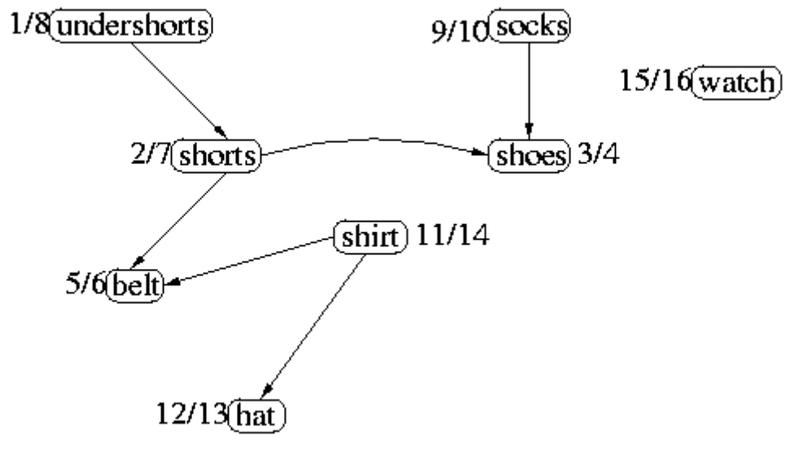
DFS_Visit(v)
Input: the node to visit, v
Postcondition: All nodes reachable from v will have been visited

```
    color[v] = gray             // Mark v as discovered
    time = time + 1
    d[v] = time
    for each u adjacent to v
        if color[u] == white
            DFS_Visit(u)        // Found a new node; explore it
    color[v] = black            // Mark v as fully explored
    time = time + 1
    f[v] = time
```

- Example: Getting dressed (borrowed from Introduction to Algorithms by Cormen, Leiserson and Rivest)
  - Getting dressed in the morning, there are certain things I have to put on before I can put something else on.
  - Represent this as a graph:



  - A vertex is everything I need to put on
  - An edge means that after I have that item on, I can put on the destination article.
  - Below is the modified DFS with finishing times



  - I can line up my clothes before I go to bed so that I know in what order to put things on: