

**6.2**

**Elementary  
Graph  
Operations**

2018/10/1 © Ren-Song Tsay, NTHU, Taiwan 29

---

---

---

---

---

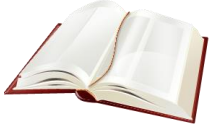
---

---

---

**Graph Operations**

- Graph traversal
  - Depth-first search
  - Breadth-first search
- Connected components
- Spanning trees



30

---

---

---

---

---

---

---

---

6.2.1 **Depth-First Search (DFS)**

- Starting from a vertex  $v$ 
  - Visit the vertex  $v \Rightarrow \text{DFS}(v)$ .
  - For each vertex  $w$  adjacent to  $v$ , if  $w$  is not visited yet, then visit  $w \Rightarrow \text{DFS}(w)$ .
  - If a vertex  $u$  is reached such that all its adjacent vertices have been visited, we go back to the last visited vertex.
- The search terminates when no unvisited vertex can be reached from any of the visited vertices.

31

---

---

---

---

---

---

---

---

### Depth-First Search (DFS)

```

graph TD
    0((0)) --- 1((1))
    0((0)) --- 2((2))
    1((1)) --- 3((3))
    1((1)) --- 4((4))
    2((2)) --- 5((5))
    2((2)) --- 6((6))
    3((3)) --- 7((7))
    4((4)) --- 7((7))
    5((5)) --- 7((7))
  
```

32

---

---

---

---

---

---

---

---

### Recursive DFS

```

void Graph::DFS(void) {
    visited = new bool[n]; // this is a data member of Graph
    fill(visited, visited+n, false);
    DFS(0); // start search at vertex 0
    delete [] visited;
}

void Graph::DFS(const int v) {
    // visit all previously unvisited vertices that are adjacent to v
    output(v);
    visited[v]=true;
    for(each vertex w adjacent to v)
        if(!visited[w]) DFS(w);
}
  
```

33

---

---

---

---

---

---

---

---

### Non-Recursive DFS

```

void Graph::DFS(int v) {
    visited = new bool[n]; // this is a data member of Graph
    fill(visited, visited+n, false);
    Stack<int> s; // declare and init a stack
    s.Push(v);
    while(!s.IsEmpty()) {
        v = s.Top(); s.Pop();
        if(!visited[v]) {
            output(v);
            visited[v]=true;
            for(each vertex w adjacent to v)
                if(!visited[w]) s.Push(w);
        }
    }
}
  
```

34

---

---

---

---

---

---

---

---

### DFS Complexity

- Adjacency matrix
  - Time to determine all adjacent vertices:  $O(n)$
  - At most  $n$  vertices are visited:  
 $O(n \cdot n) = O(n^2)$
- Adjacency lists
  - There are  $n + 2e$  chain nodes
  - Each node in the adjacency list is examined at most once. Time complexity =  $O(e)$

35

---

---

---

---

---

---

---

---

6.2.2

### Breadth-First Search (BFS)

- Starting from a vertex  $v$ 
  - Visit the vertex  $v$ .
  - Visit all unvisited vertices adjacent to  $v$ .
  - Unvisited vertices adjacent to these newly visited vertices are then visited and so on...

36

---

---

---

---

---

---

---

---

### Breadth-First Search (BFS)

37

---

---

---

---

---

---

---

---

### BFS: Implementation

```

void Graph::BFS(int v){
    visited = new bool[n]; // this is a data member of Graph
    fill(visited, visited+n, false);
    Queue<int> q; // declare and init a queue
    q.Push(v);
    visited[v]=true;
    while(!q.IsEmpty()){
        v = q.Front(); q.Pop();
        output(v);
        for(each vertex w adjacent to v){
            if(!visited[w]){
                q.Push(w);
                visited[w]=true;
            }
        }
    }
    delete [] visited;
}
    
```

Time complexity is the same as DFS

---

---

---

---

---

---

---

---

### 6.2.3 Connected Components

- How to determine whether a graph is connected or not?
  - Call DFS or BFS once and check if there is any unvisited vertices, if Yes, then the graph is not connected.
- How to identify connected components
  - Make a repeated calls to DFS or BFS.
  - Each call will output a connected component.
  - Start next call at an unvisited vertex.

---

---

---

---

---

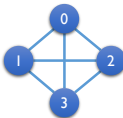
---

---

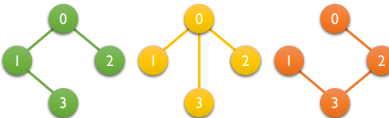
---

### 6.2.4 Spanning Trees

- Definition: Any tree consists of solely of edges in  $E(G)$  and including all vertices of  $V(G)$ .
- Number of tree edges is  $n - 1$ .
- Add a non-tree edge will create a **cycle**.



Complete graph



Possible spanning trees

---

---

---

---

---

---

---

---

### DFS Spanning Tree

- Tree edges are those edges met during the traversal.

The diagram shows a graph with nodes 0 through 7. Node 0 is the root. Node 1 is the left child of 0, and node 2 is the right child. Node 3 is the left child of 1, node 4 is the right child of 1, node 5 is the left child of 2, and node 6 is the right child of 2. Node 7 is the left child of 3. A traversal path is shown in yellow boxes: 0 → 1 → 3 → 7 → 4 → 5 → 2 → 6. A green arrow labeled 'DFS' points from the initial graph to the resulting spanning tree. The spanning tree edges are (0,1), (0,2), (1,3), (1,4), (2,5), (2,6), and (3,7).

---

---

---

---

---

---

---

---

### BFS Spanning Tree

- Tree edges are those edges met during the traversal.

The diagram shows a graph with nodes 0 through 7. Node 0 is the root. Node 1 is the left child of 0, and node 2 is the right child. Node 3 is the left child of 1, node 4 is the right child of 1, node 5 is the left child of 2, and node 6 is the right child of 2. Node 7 is the left child of 3. A traversal path is shown in yellow boxes: 0 → 1 → 2 → 3 → 4 → 5 → 6 → 7. A green arrow labeled 'BFS' points from the initial graph to the resulting spanning tree. The spanning tree edges are (0,1), (0,2), (1,3), (1,4), (2,5), (2,6), and (3,7).

---

---

---

---

---

---

---

---