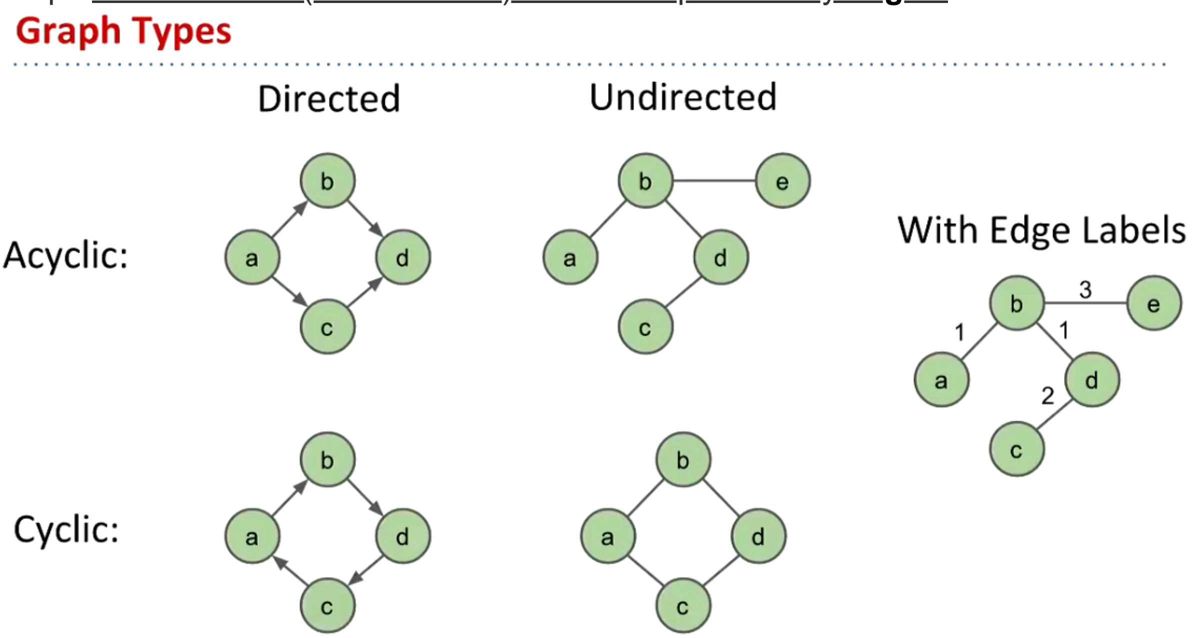
CS61B Week 10 Graph & Graph Traversal

Lec26.Graphs

Part1.Introduction

1. Graph:A set of **nodes**(a.k.a vertices) connected pairwise by **edges**.

   Vertices with an edge between are adjacent(邻居).*Vertices or edges may have labels(or weights)*
2. A **path** is a sequence of vertices connected by edges.

   A **cycle** is a path whose first and last vertices are the same.A graph with a cycle is 'cyclic'
3. Two vertices are **connected** if there is a path between them.If all verties are connected,we say the graph is connected.
4. **degree**

   How many nodes are connected to a node?

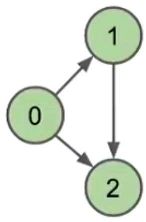
Part2.Graph Implementation

```
public class Graph{
    public Graph(int V); //Create empty graph with v vertices
    public void addEdge(int v, int w);  //add an edge v-w
    Iterable<Integer> adj(int v); //vertices adjacent to v
    int V(); //number of vertices
    int E(); //number of edges

    public static int degree(Graph G, int v){
       int degree = 0;
       for (int w : G.agj(v)){
        degree += 1;
       }
       return degree;
    }
}
```

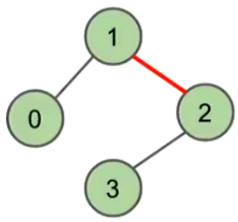
Representation1: Adjacency Matrix(邻接矩阵)

s \ t	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0



For undirected graph: Each edge is represented twice in the matrix. Simplicity at the expense of space.

v \ w	0	1	2	3
0	0	1	0	0
1	1	0	**1**	0
2	0	**1**	0	1
3	0	0	1	0



Time complexity of print edge

What is the order of growth of the running time of the following code if the graph uses an adjacency-matrix representation, where V is the number of vertices, and E is the total number of edges?

A. Θ(V)
B. Θ(V + E)
C. Θ(V^2)
D. Θ(V*E)

```
for (int v = 0; v < G.V(); v++) {
    for (int w : G.adj(v)) {
        System.out.println(v + "-" + w);
    }
}
```

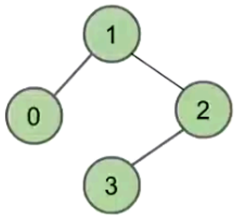
What is the runtime of the for-each?

- Θ(V).

How many times is the for-each run?

- V times.

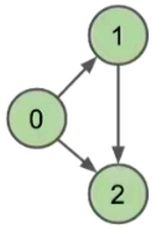
	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0



Representation2:Edges Sets:Collection of all edges

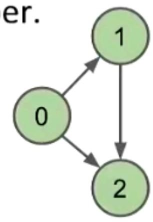
- Example: HashSet<Edge>, where each Edge is a pair of ints.

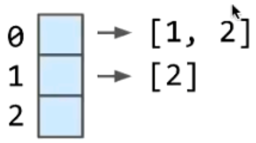
```
{(0, 1), (0, 2), (1, 2)}
```



Representation3:Adjacnecy Lists(邻接链表/列表)

- Common approach: Maintain array of lists indexed by vertex number.
- Most popular approach for representing graphs.





Time complexity of print edge

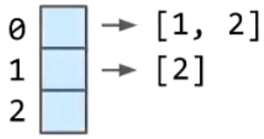
What is the order of growth of the running time of the following code if the graph uses an ***adjacency-list*** representation, where V is the number of vertices and E is the total number of edges?

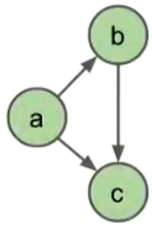
A. $\Theta(V)$
B. $\Theta(V + E)$
C. $\Theta(V^2)$
D. $\Theta(V^*E)$

```
for (int v = 0; v < G.V(); v++) {
    for (int w : G.adj(v)) {
        System.out.println(v + "-" + w);
    }
}
```

Best case: $\Theta(V)$ Worst case: $\Theta(V^2)$

All cases: $\Theta(V + E)$





How to interpret: No matter what "shape" of increasingly complex graphs we generate, as V and E grow, the runtime will always grow exactly as $\Theta(V + E)$.

- Example shape 1: Very sparse graph where E grows very slowly, e.g. every vertex is connected to its square: 2 - 4, 3 - 9, 4 - 16, 5 - 25, etc.
  - E is $\Theta(\mathrm{sqrt}(V))$. Runtime is $\Theta(V + \mathrm{sqrt}(V))$, which is just $\Theta(V)$.
- Example shape 2: Very dense graph where E grows very quickly, e.g. every vertex connected to every other.
  - E is $\Theta(V^2)$. Runtime is $\Theta(V + V^2)$, which is just $\Theta(V^2)$.

```
public class Graph{
    private final int V;
    private List<Integer>[] adj;

    public Graph(int V){
        this.V=V;
        adj = (List<Integer>[]) new ArrayList[V];
        for (int v = 0; v < V; v++){
            adj[v] = new ArrayList <Integer>();
        }
    }

    public void addEdge(int v, int w){
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterator <Integer> adj(int v){
        return adj[v];
    }
}
```

Runtime Comparasion between different representation:

idea	addEdge(s, t)	for(w : adj(v))	printgraph()	hasEdge(s, t)	space used
adjacency matrix	$\Theta(1)$	$\Theta(V)$	$\Theta(V^2)$	$\Theta(1)$	$\Theta(V^2)$
list of edges	$\Theta(1)$	$\Theta(E)$	$\Theta(E)$	$\Theta(E)$	$\Theta(E)$
adjacency list	$\Theta(1)$	$\Theta(1)$ to $\Theta(V)$	$\Theta(V+E)$	$\Theta(\text{degree}(v))$	$\Theta(E+V)$

Lec27.Graph Traversal

Part1.Depth First Traversal

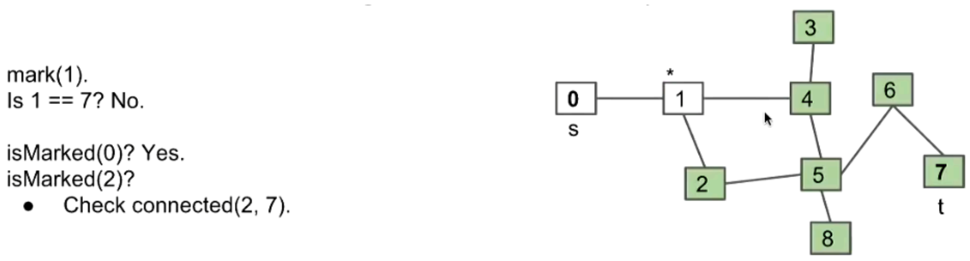
1. The Connected(has path to) Function

   Goal(The s-t problem):

   Search for a path from s to t, but **visit each vertex at most once**.To do this, we can **mark each vertex as we search**.

   Resulting algorithm for connected(s,t) is as follows:

   - Mark s
   - Does s==t? If so ,return true
   - Check all of s's unmarked neighbors for connectivity to t.



2. Depth First Traversal(Search) Implementation

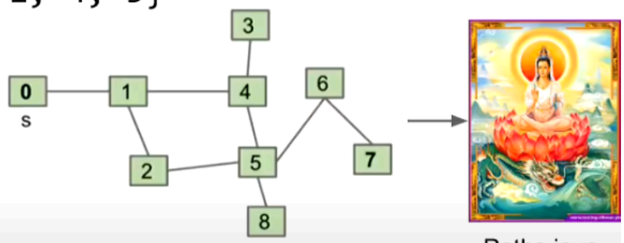
   > Common design pattern in graph algorithms:**Decouple(分离) type from processing algorithm.**

   - Create a graph object
   - Pass the graph to a graph-processing method(or constructor) in a client class
   - Query the client class for imformation

```
public class Path{
  public Path(Graph G, int s);
  //Find all paths from G ,s is the source node.
  boolean hasPathTo(int v);
  //is there a path from s to v?
  Iterable<Integer> pathTo(int v);
  //path from s to v(if any)
}
```

Start by calling: `Paths P = new Paths(G, 0);`

- `P.hasPathTo(3); //returns true`
- `P.pathTo(3); //returns {0, 1, 4, 3}`



Paths.java

Implementing Paths With Depth Forst Search

To visit a vertex v,we need to mark the vertex v and recursively visit all unmarked vertices adjacent to v.

Data Structure needed:

- boolean[] marked;
  //If the vertex is marked?
- int[] edgeTo;
  //to record the path we have gone

```java
public class DepthFirstPaths{
    private boolean[] marked;
    /*marked[v] is true if v connects to s/
    private int[] edgeTo;
    /*edgeTo[v] is previous vertex on path from s to v*/
    private int s;
    /*the source node*/

    public DepthFirstPaths(){
        ...
        dfs(G, s);
    }

    private void dfs(Graph G, int v){
        marked[v] = true;
        for(int w : G.adj(v)){
            if(!marked[w]){
                edgeTo[w]=v;
                dfs(G, w);
            }
        }
    }

    public boolean hasPathTo(int v){
        return marked[v];
    }
}
```

Goal: Find a path from s to every other reachable vertex, visiting each vertex at most once. dfs(v) is as follows:

- Mark v.
- For each unmarked adjacent vertex w:
    - set edgeTo[w] = v.
    - dfs(w)

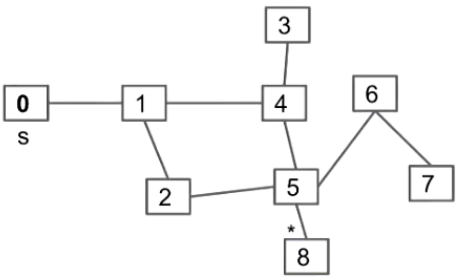
#	marked	edgeTo
0	T	-
1	T	0
2	T	1
3	T	4
4	T	5
5	T	2
6	T	5
7	T	6
8	T	5

dfs(8):
  mark(8)

  isMarked(5)? Yes.

  No more children, so return.

Order of dfs calls: 012543678



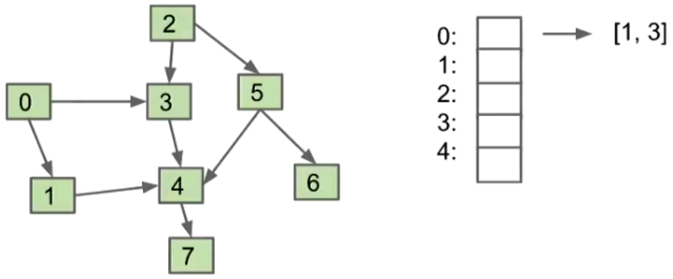
Order of dfs returns: 34768

- Runtime: $\Theta(V+E)$ each vertex is visited once,Each visit costs constant time
- Space: $\Theta(V)$ Call stack depth is at most V

3. Types of DFS

- pre-order traversal

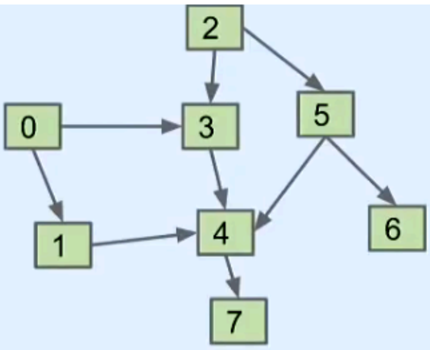
- in-order traversal
- post-order traversal



What is the DFS postorder of this graph starting at 0? Assume items appear in adjacency lists in increasing order. DFS postorder is the order of *returns* from DFS.

A. 0 1 3 4 7
B. 7 4 3 1 0
C. 7 4 1 0 3
D. 7 4 1 3 0
E. 0 1 4 7 3

Part2.Topological Sort



Suppose we have tasks 0 through 7, where an arrow from v to w indicates that v must happen before w.

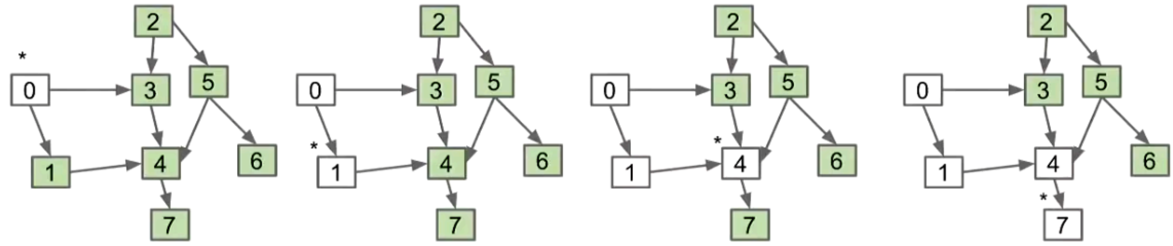
- What algorithm do we use to find a valid ordering for these tasks?
- Valid orderings include: [0, 2, 1, 3, 5, 4, 7, 6], [2, 0, 3, 5, 1, 4, 6, 7], …

Any suggestions on where we'd start?

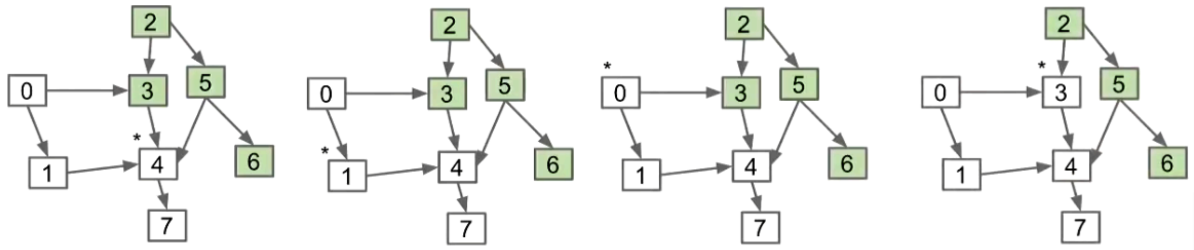
Perform a DFS traversal from every from every vertex with indegree 0, NOT clearing markings in between traversals

- Record DFS post order in a list
- Topological ordering is given by the reverse of that list(reverse postorder)

Topological Sort (Demo 1/2)

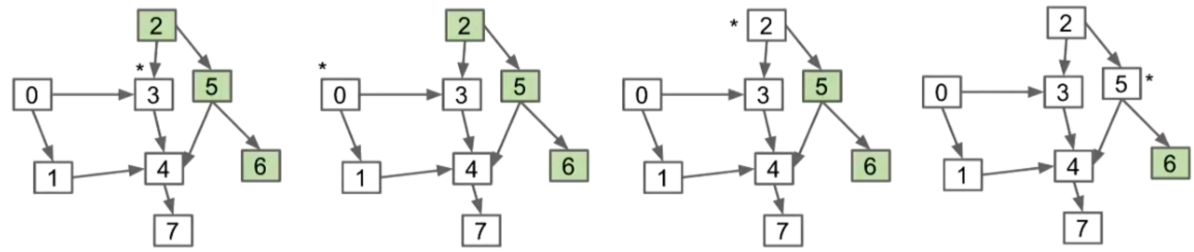


Postorder: [] | Postorder: [] | Postorder: [] | Postorder: [7]

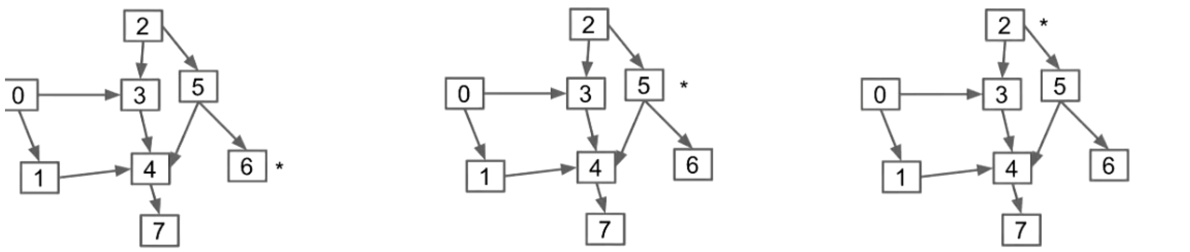


Postorder: [7, 4] | Postorder: [7, 4, 1] | Postorder: [7, 4, 1] | Postorder: [7, 4, 1, 3]

Topological Sort (Demo 2/2)



Postorder: [7, 4, 1, 3] | Postorder: [7, 4, 1, 3, 0] | Postorder: [7, 4, 1, 3, 0] | Postorder: [7, 4, 1, 3, 0]



Postorder: [7, 4, 1, 3, 0, 6] | Postorder: [7, 4, 1, 3, 0, 6, 5] | Postorder: [7, 4, 1, 3, 0, 6, 5, 2]

Topological Sort Implementation

```java
public class DepthFirstOrder {
    private boolean[] marked;
    private Stack<Integer> reversePostorder;
    public DepthFirstOrder(Digraph G) {
    reversePostorder = new Stack<Integer>();
    marked = new boolean[G.V()];
    for (int v = 0; v < G.V(); v++) {
        if (!marked[v]) { dfs(G, v); }
    }
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v)) {
            if (!marked[w]) { dfs(G, w); }
        }
        reversePostorder.push(v);
    }
    public Iterable<Integer> reversePostorder()
    { return reversePostorder; }
}
```

Textbook implementation (shown here) uses a stack instead of a creating a list and then reversing it.

Create empty stack.

Perform DFS of all unmarked vertices.

After each DFS is done, 'visit' vertex by putting on a stack.

Runtime: $\Theta(V + E)$
Space:$\Theta(V)$

Part3.Breadth First Search

(a.k.a Level order traversal)

1. Breadth First Search
    - Initialize a queue(we call this the fringe 带) with a starting vertex 's' and mark that vertex
    - Repeat until queue is empty:
        - **Remove vertex v fom the queue**
        - **Add to the queue any unmarked vertices adjacent to v and mark them,set its edgeTo = v**