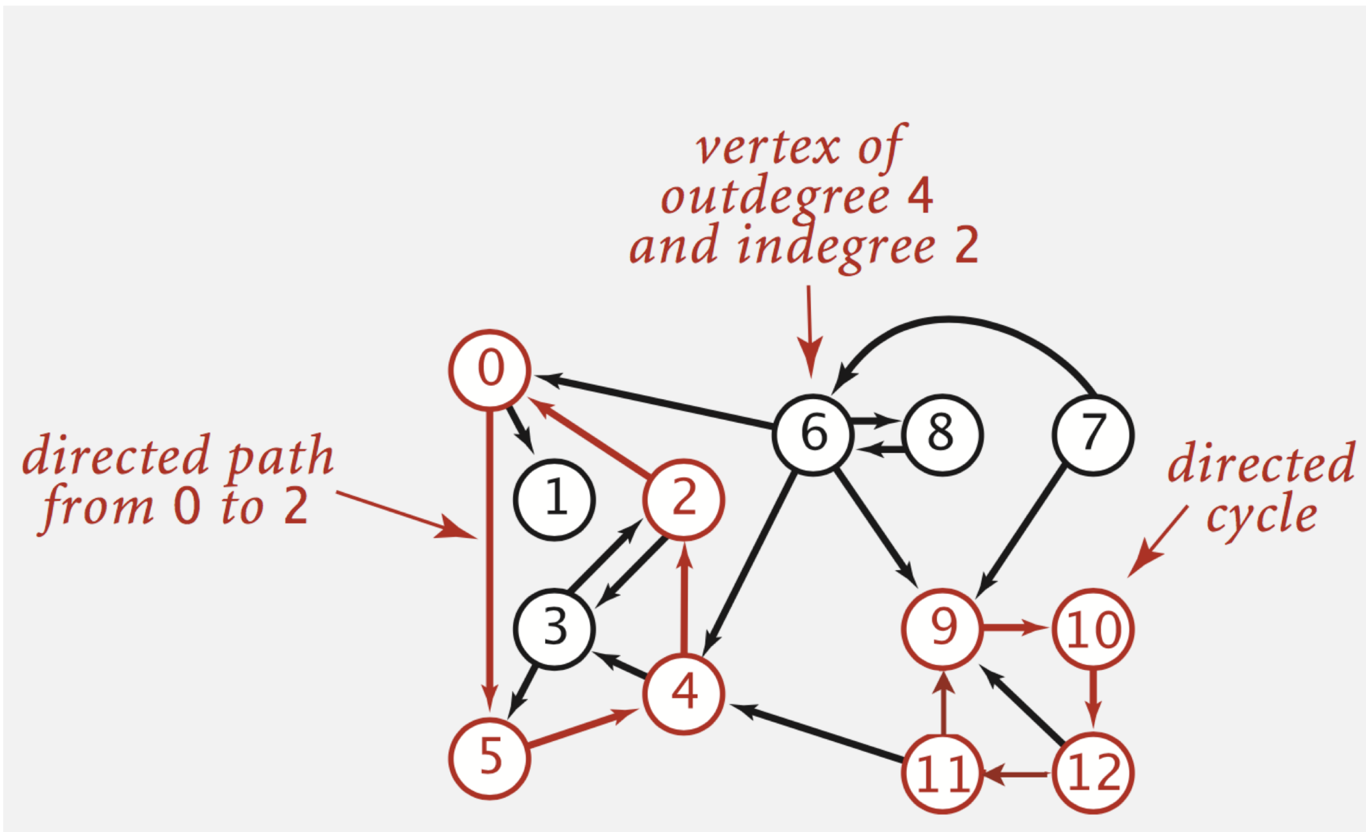
CMSC 132: Object-Oriented Programming II

DIRECTED GRAPHS

Graphs slides are modified from COS 126 slides of Dr. Robert Sedgewick.

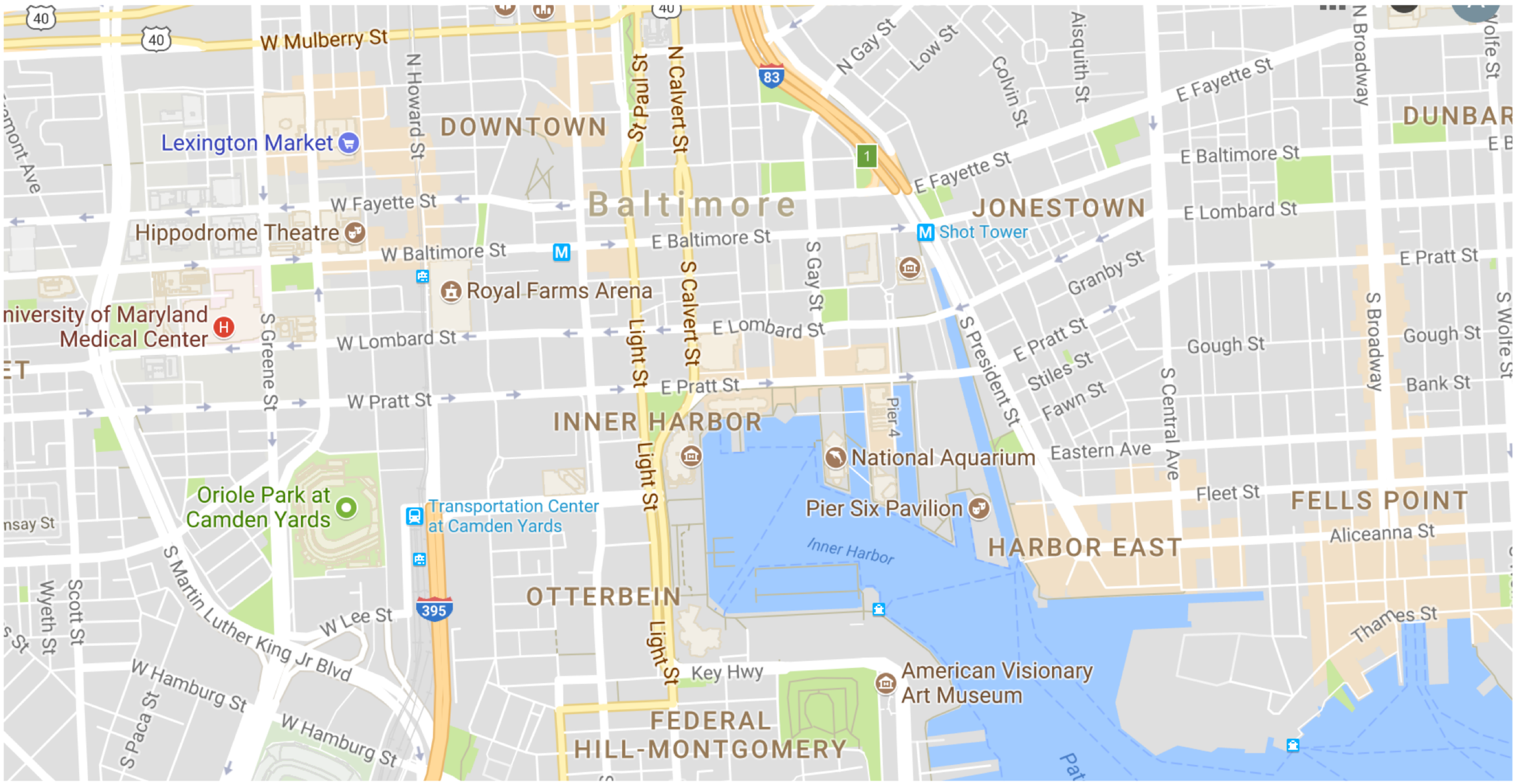
Directed graphs

- Digraph
  - Set of vertices connected pairwise by directed edges.



Road network

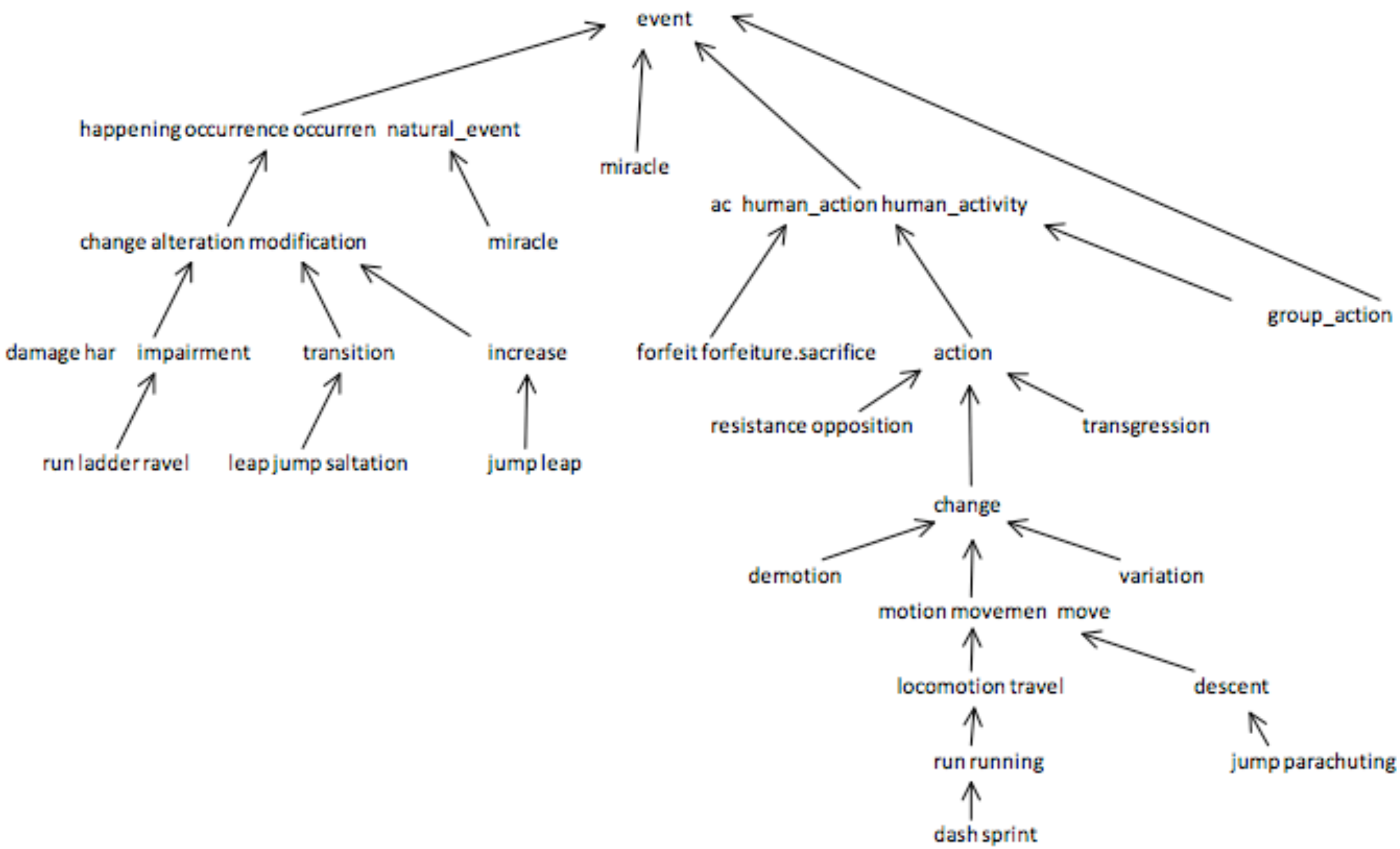
Vertex = intersection; edge = one-way street.



Baltimore inner harbor

WordNet graph

Vertex = synset; edge = hypernym relationship.



Digraph applications

digraph	vertex	edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Some digraph problems

- Path:
  - Is there a directed path from *s* to *t* ?
- Shortest path:
  - What is the shortest directed path from *s* to *t* ?
- Topological sort:
  - Can you draw a digraph so that all edges point upwards?
- Strong connectivity:
  - Is there a directed path between all pairs of vertices?
- Transitive closure:
  - For which vertices *v* and *w* is there a path from *v* to *w* ?
- PageRank:
  - What is the importance of a web page?

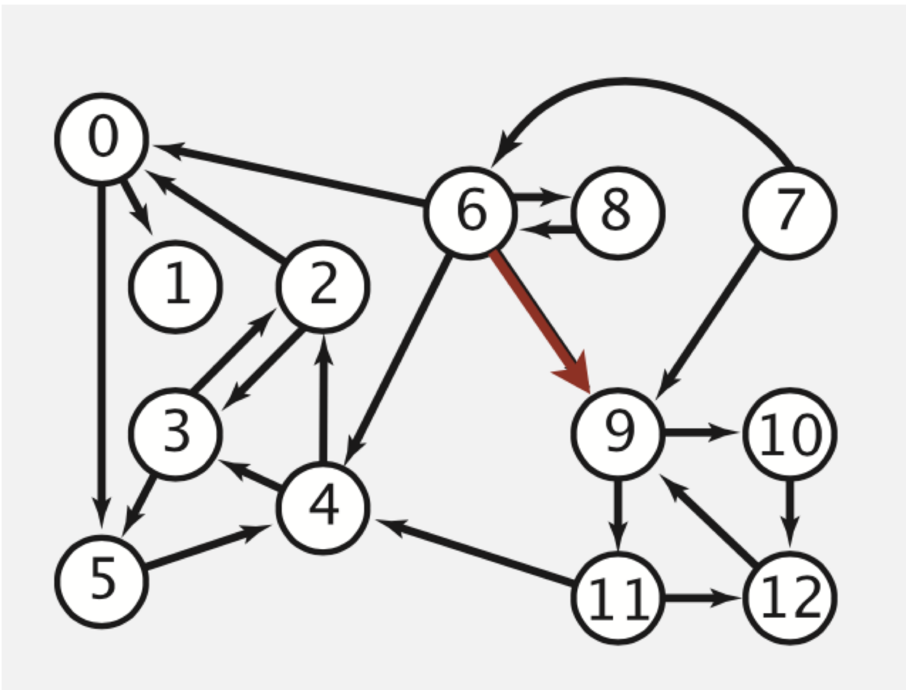
Digraph Implementation

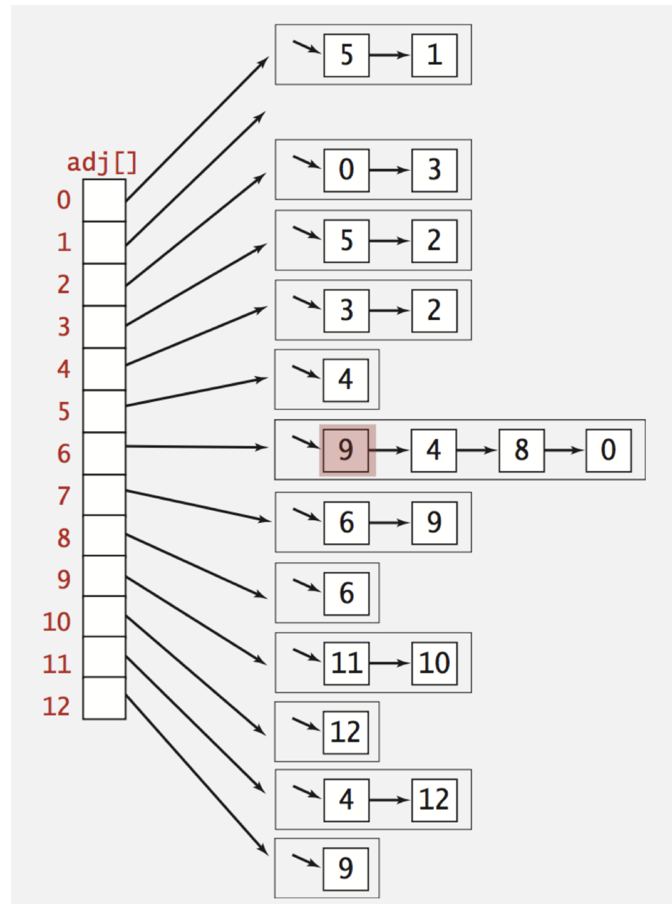
public class Digraph

	Digraph(int V)	*create an empty digraph with V vertices*
	Digraph(In in)	*create a digraph from input stream*
void	addEdge(int v, int w)	*add a directed edge v→w*
Iterable<Integer>	adj(int v)	*vertices pointing from v*
int	V()	*number of vertices*
int	E()	*number of edges*
Digraph	reverse()	*reverse of this digraph*
String	toString()	*string representation*

Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.





Adjacency-lists digraph implementation

```
public class Graph {
  private final int V;
  private final Bag<Integer>[] adj;        adjacency lists
  public Graph(int V) {
    this.V = V;
    adj = (Bag<Integer>[]) new Bag[V];        create empty graph with
    for (int v = 0; v < V; v++)                V vertices
        adj[v] = new Bag<Integer>();
  }
  public void addEdge(int v, int w) {
      adj[v].add(w);
  }
  public Iterable<Integer> adj(int v) {      iterator for vertices
       return adj[v];                        pointing from v
  }
}
```

Digraph representation

Comparisons of three different representations:

representation	space	insert edge from v to w	edge from v to w?	iterate over vertices pointing from v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 †	1	V
adjacency lists	$E + V$	1	outdegree(v)	outdegree(v)

† disallows parallel edges

Depth-first search in digraphs

- Same method as for undirected graphs.
  - Every undirected graph is a digraph (with edges in both directions).
  - DFS is a digraph algorithm.

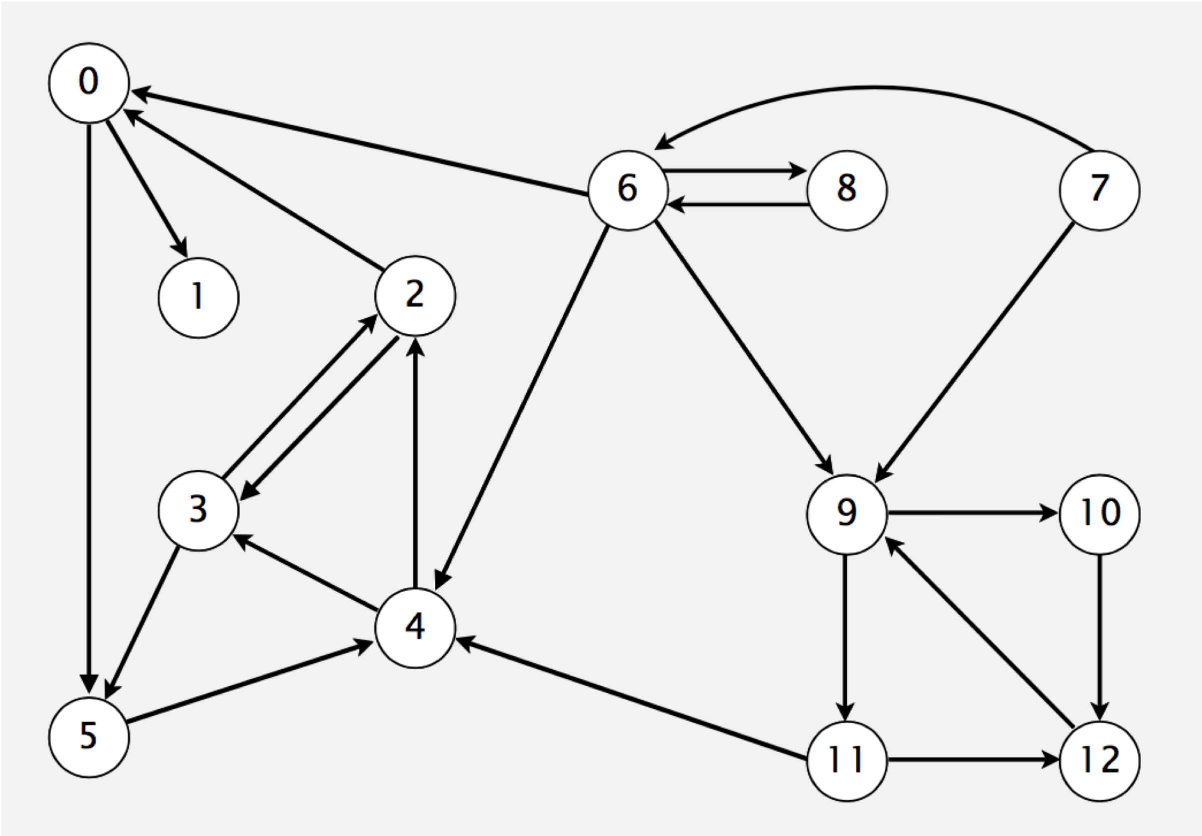
DFS (to visit a vertex v)
    Mark v as visited.
    Recursively visit all unmarked vertices w pointing from v.

Depth-first search demo

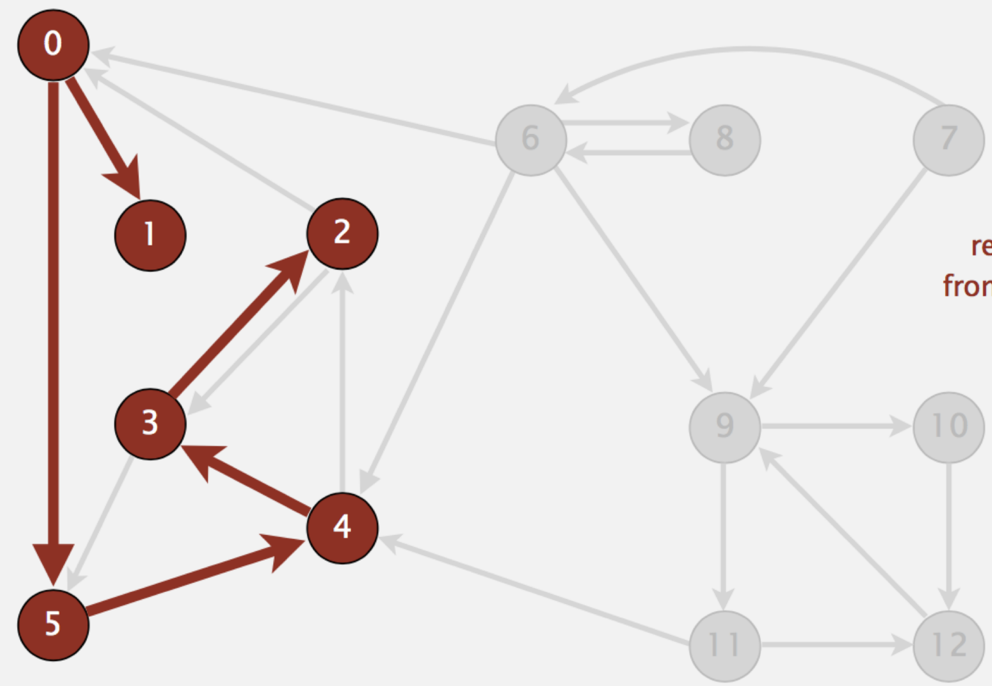
To visit a vertex *v* :

Mark vertex *v* as visited.

Recursively visit all unmrked vertices pointing from *v*.



Depth-first search demo



reachable from vertex 0

v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

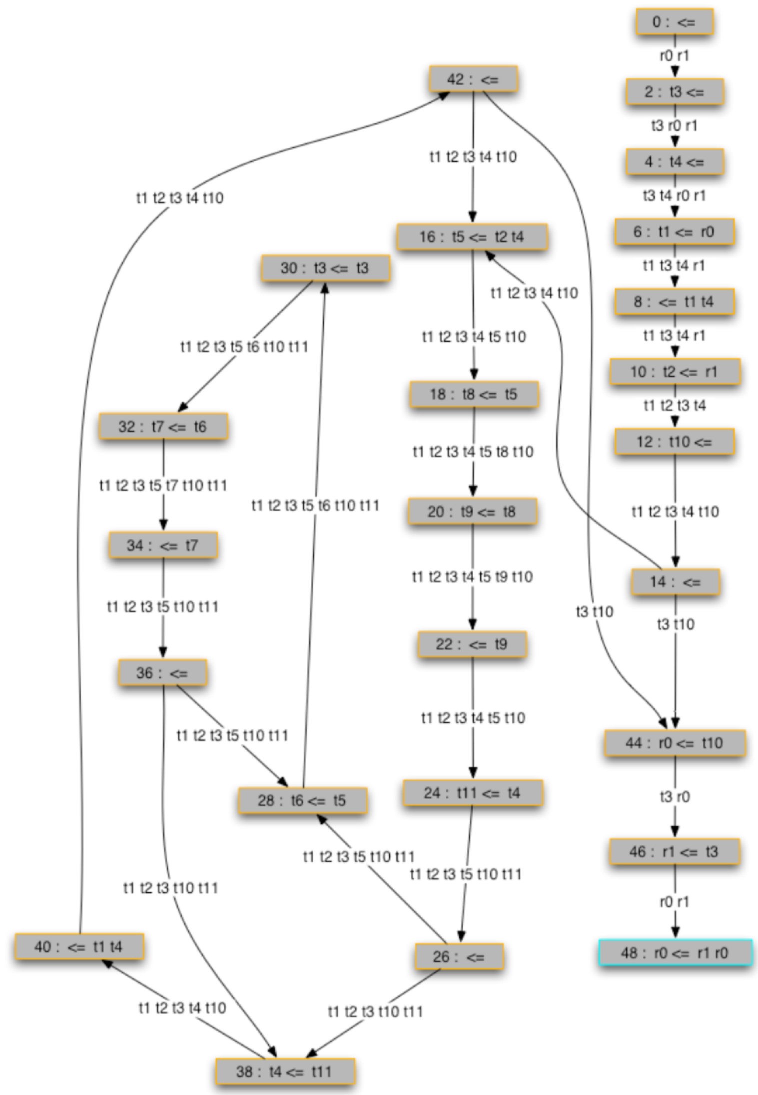
Depth-first search Implementation

Code for directed graphs identical to undirected one.

```
public class DirectedDFS {
  private boolean[] marked;
  public DirectedDFS(Digraph G, int s) {
    marked = new boolean[G.V()];
    dfs(G, s);
  }
  private void dfs(Digraph G, int v) {
     marked[v] = true;
     for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
  }
  public boolean visited(int v) {
      return marked[v];
  }
}
```

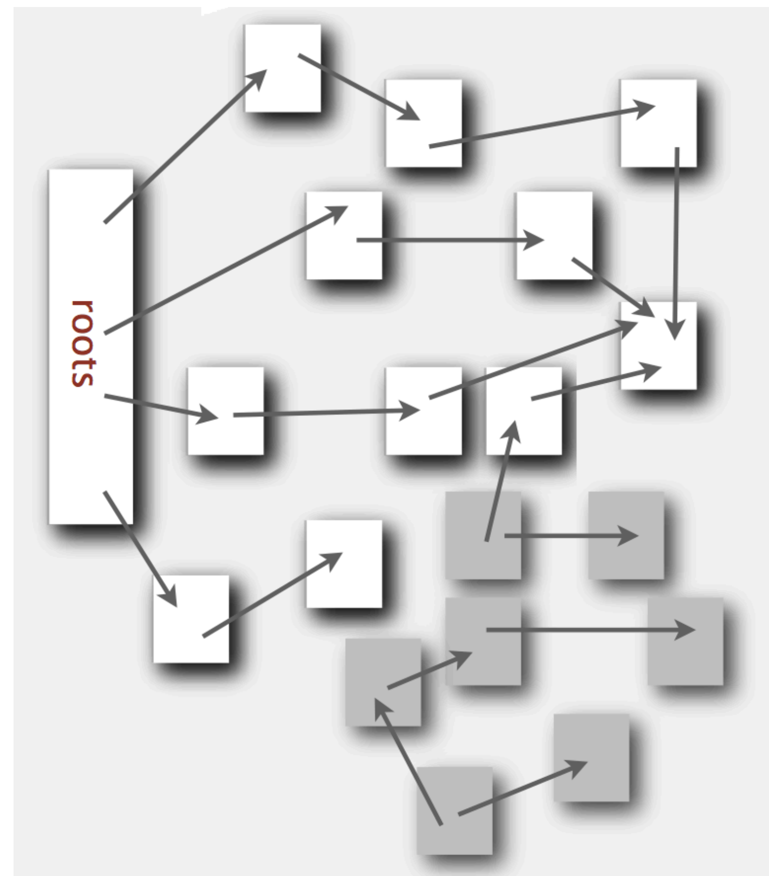
Reachability application: program control-flow analysis

- Every program is a digraph.
  - Vertex = basic block of instructions (straight-line program).
  - Edge = jump.
- Dead-code elimination.
  - Find (and remove) unreachable code.



Reachability application: mark-sweep garbage collector

- Every data structure is a digraph.
  - Vertex = object.
  - Edge = reference.
- Roots:
  - Objects known to be directly accessible by program (e.g., stack).
- Reachable objects:
  - Objects indirectly accessible by program (starting at a root and following a chain of pointers).



Breadth-first search in digraphs

Same method as for undirected graphs. Every undirected graph is a digraph (with edges in both directions). BFS is a digraph algorithm.

BFS (from source vertex s)
Put s onto a FIFO queue, and mark s as visited.
Repeat until the queue is empty:
- remove the least recently added vertex v
- for each unmarked vertex pointing from v:
add to queue and mark as visited.

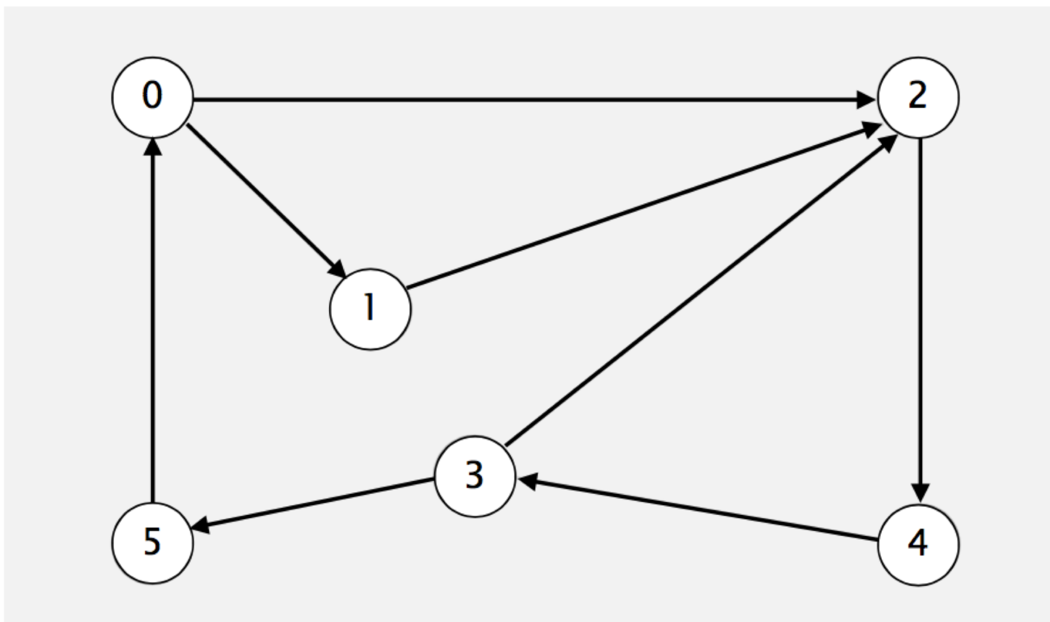
Proposition. BFS computes shortest paths (fewest number of edges) from *s* to all other vertices in a digraph in time proportional to $E + V$.

Directed breadth-first search demo

Repeat until queue is empty:

Remove vertex *v* from queue.

Add to queue all unmarked vertices pointing from *v* and mark them.

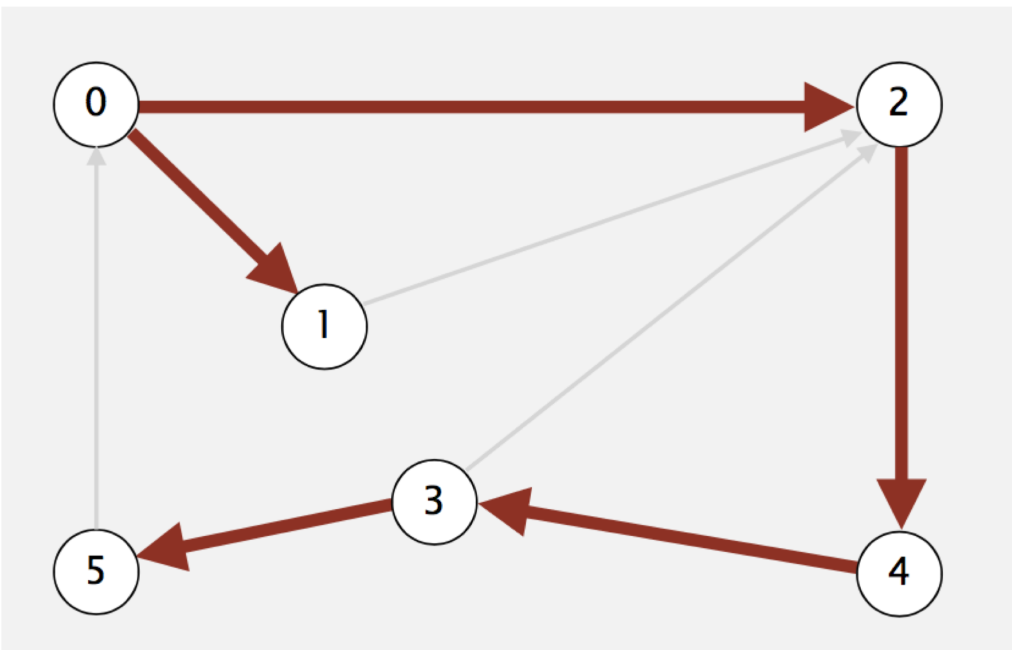


Directed breadth-first search demo

Repeat until queue is empty:

Remove vertex *v* from queue.

Add to queue all unmarked vertices pointing from *v* and mark them.

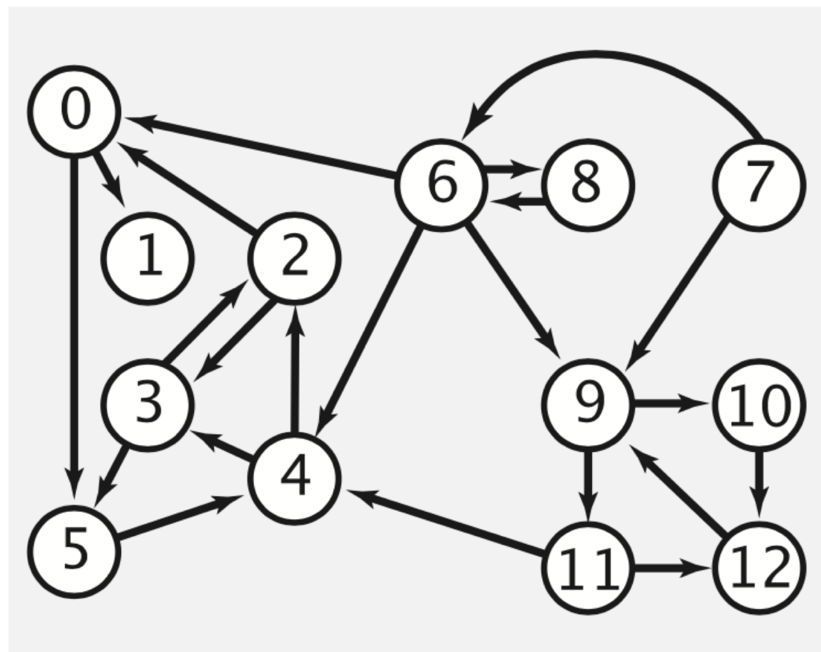


v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

Multiple-source shortest paths

- Given a digraph and a set of source vertices, find shortest path from any vertex in the set to each other vertex.
- Use BFS, but initialize by enqueuing all source vertices

Example:
S = {1, 7, 10 }.
Shortest path to 4 is 7→6→4. 1
Shortest path to 5 is 7→6→0→5
Shortest path to 12 is 10→12.

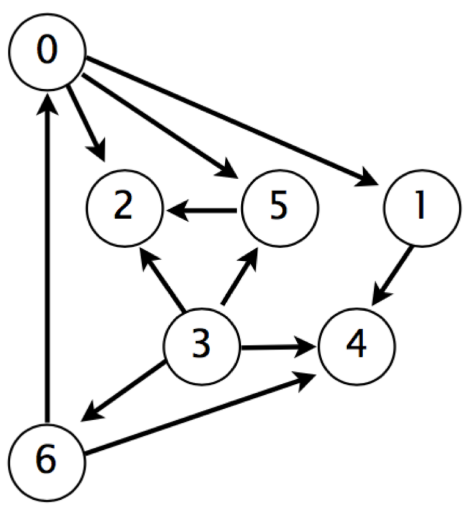


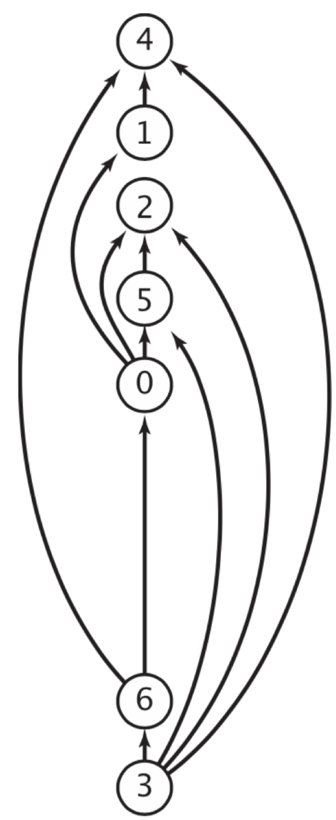
Topological Sort

Precedence scheduling

- Goal:
  - Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?
- Digraph model:
  - vertex = task;
  - edge = precedence constraint.

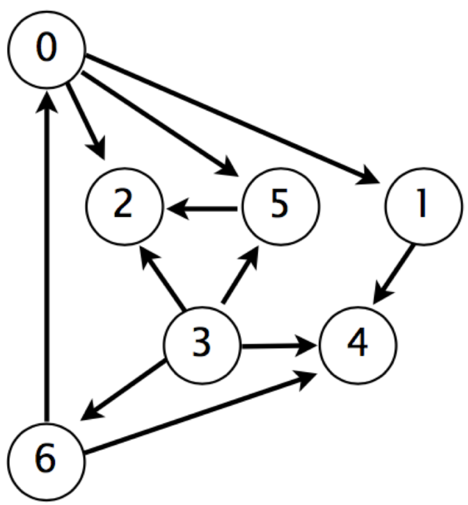
0.CMSC216
1.CMSC330
2.CMSC351
3.CMSC131
4.CMSC420
5.CMSC250
6.CMSC132

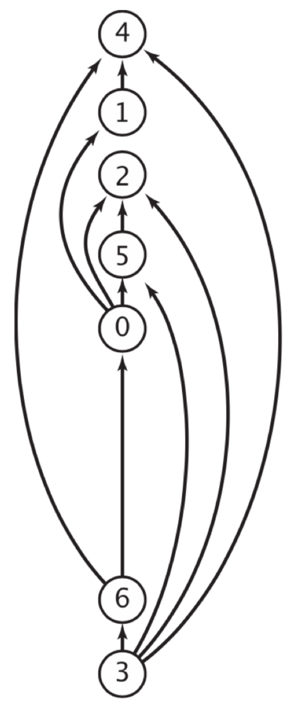




Topological sort

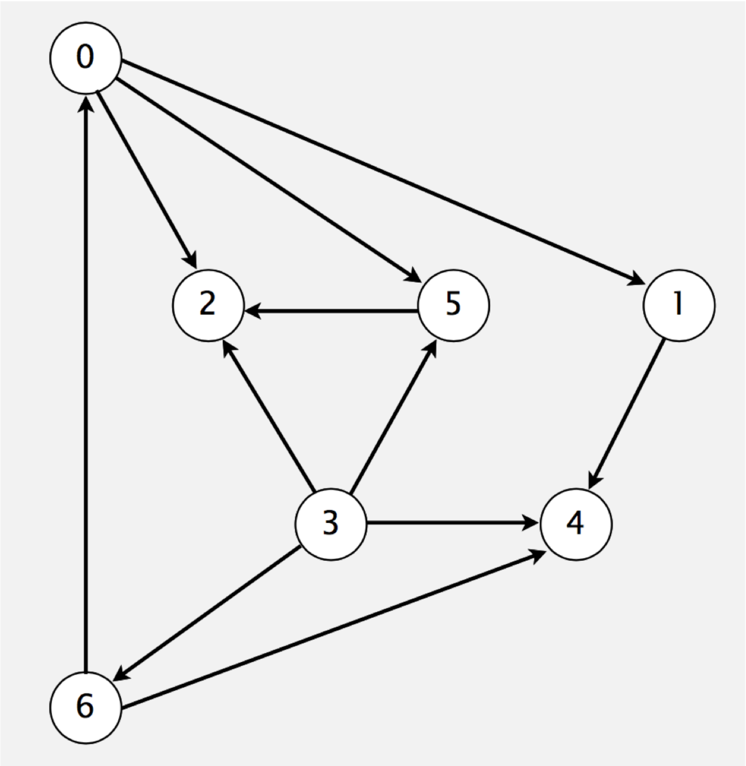
- DAG:
  - Directed acyclic graph.
- Topological sort:
  - Redraw DAG so all edges point upwards.





Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder



postorder

4,1,2,5,0,6,3

topological order

3,6,0,5,2,1,4

Depth-first search order

```
public class DepthFirstOrder {
  private boolean[] marked;
  private Stack<Integer> reversePost;
  public DepthFirstOrder(Digraph G) {
     reversePost = 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);
     reversePost.push(v);
  }
public Iterable<Integer> reversePost() {
  return reversePost;
}
}
```

Topological sort

- **Kahn's algorithm**
  - First described by Kahn (1962),

1. find a vertex which has no incoming edges
2. insert it into a set S;   at least one such vertex must exist in a non-empty acyclic graph.

2. Remove outgoing edges from that vertex, and repeat 1