ITP20001/ECE20010 Data Structures

Chapter 6

- *Introduction*
- *Graph API*
- *Elementary Graph Operations*
  - ***DFS: Depth first search***
  - *BFS: Breadth first search*
  - *CC: Connected Components*

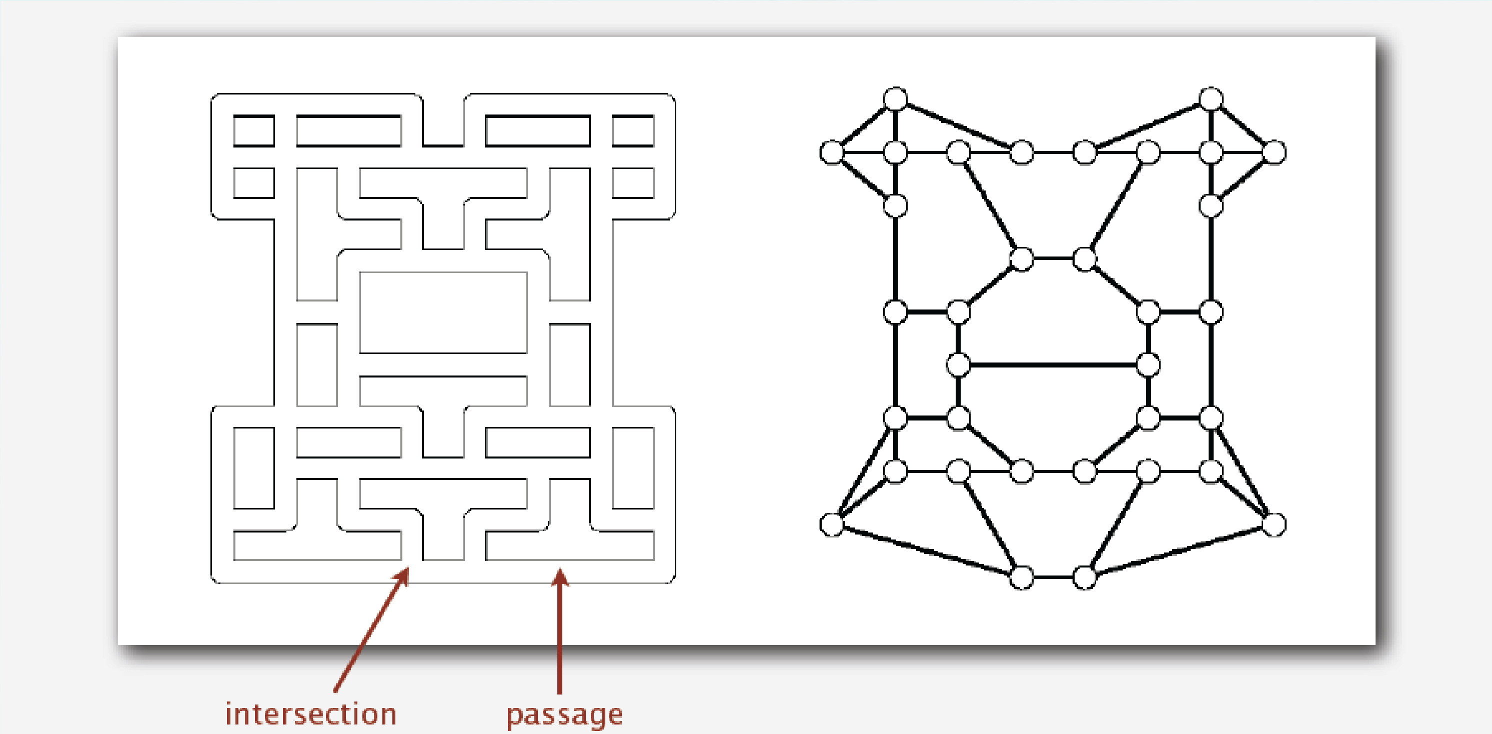
Major references:
1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University

Depth first search

Algorithm:

- Vertex = intersection
- Edge = passage



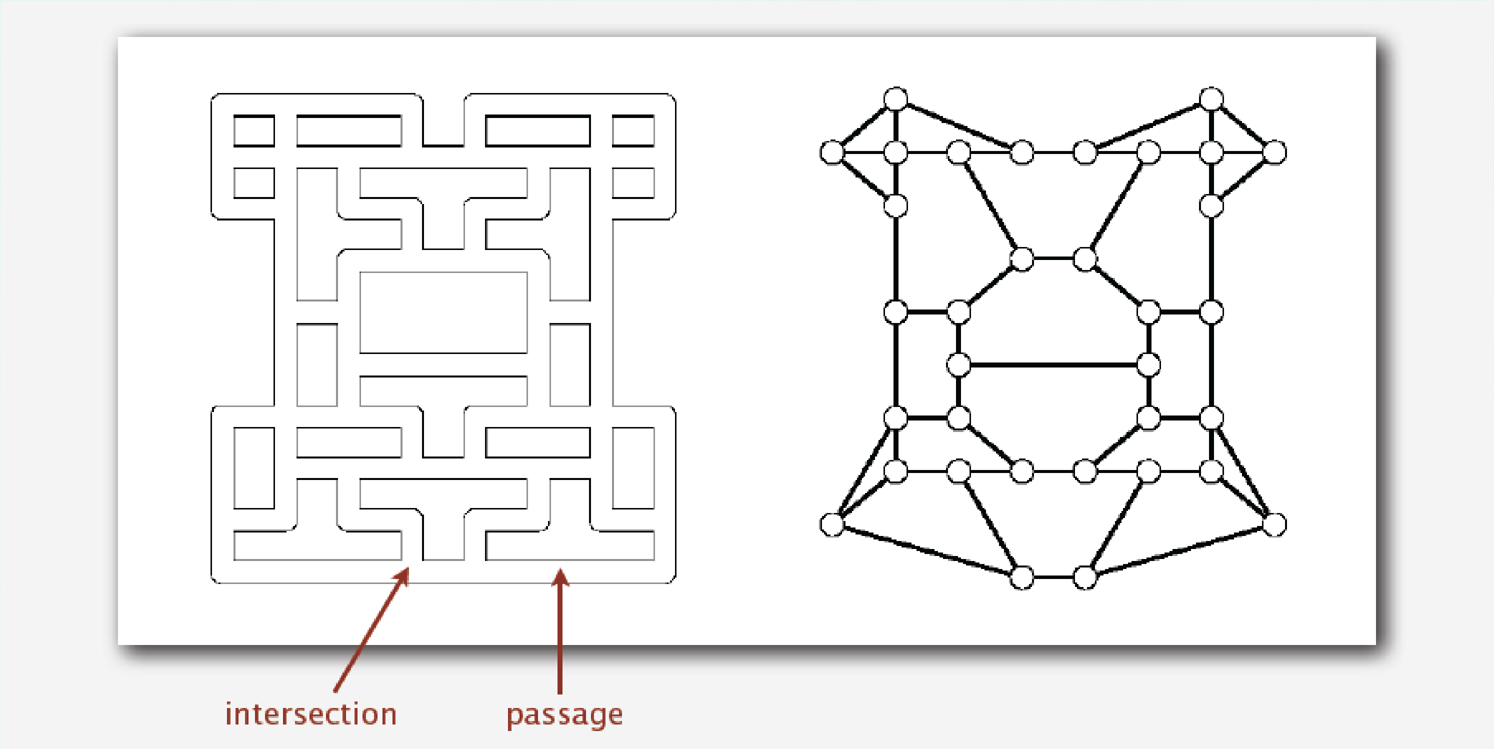
Maze Goal: Explore every intersection in the maze.

Depth first search

Algorithm:

- Vertex = intersection
- Edge = passage

pacman

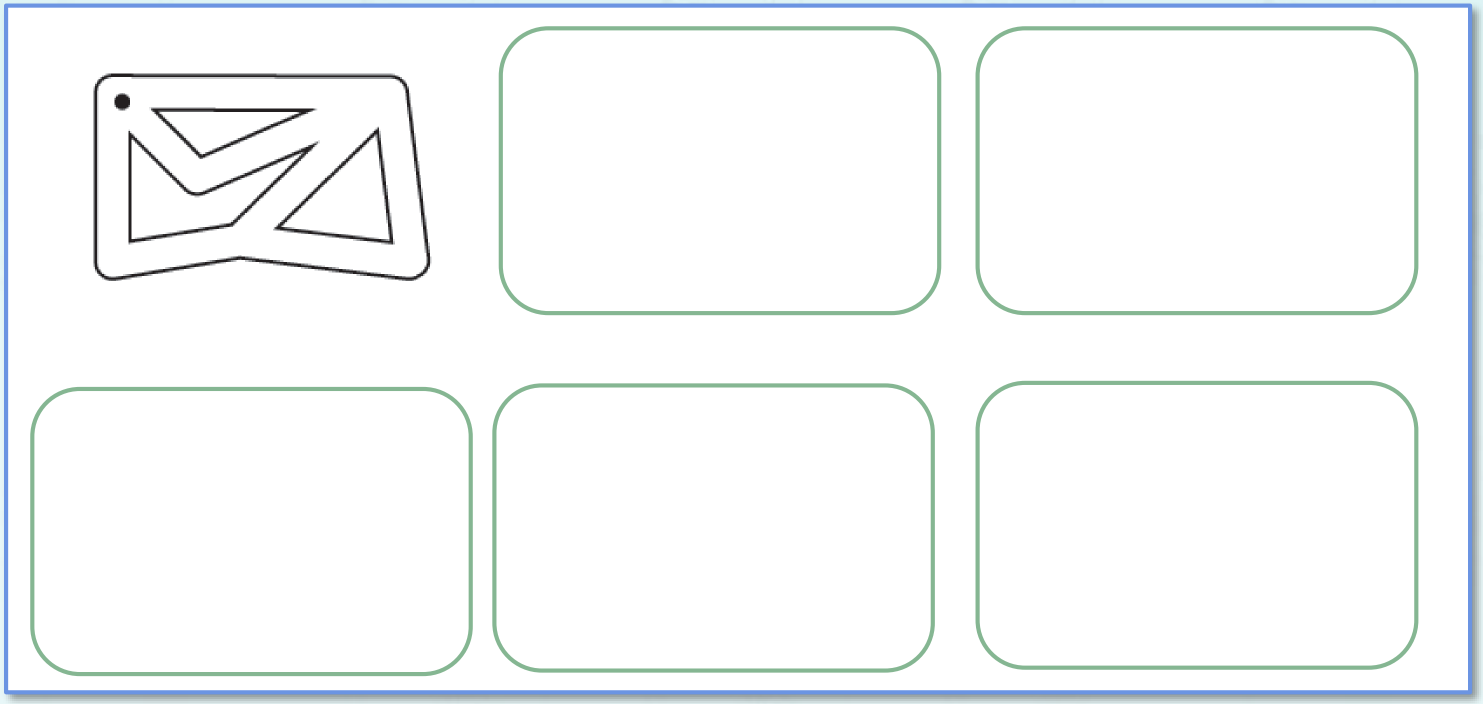


Maze Goal: Explore every intersection in the maze.

Depth first search

Maze graph:

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options



Maze Goal: Explore every intersection in the maze.

Good Visualization: https://www.cs.usfca.edu/~galles/visualization/DFS.html

Depth first search

Maze graph:

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options

Theseus, a hero of Greek mythology, is best known for slaying a monster called the Minotaur. When Theseus entered the Labyrinth where the Minotaur lived, he took a ball of yarn to unwind and mark his route. Once he found the Minotaur and killed it, Theseus used the string to find his way out of the maze.

Read more: http://www.mythencyclopedia.com/Sp-Tl/Theseus.html#ixzz30wFO30fe

Maze Goal: Explore every intersection in the maze.

Depth-first search

Maze graph:

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options

Shannon and his famous electromechanical mouse *Theseus* (named after Theseus from Greek mythology) which he tried to have solve the maze in one of the first experiments in artificial intelligence.

The Las Vegas connection: Shannon and his wife Betty also used to go on weekends to Las Vegas with MIT mathematician Ed Thorp, and made very successful forays in blackjack using game theory.

Maze Goal: Explore every intersection in the maze.

Design pattern for graph processing

Design pattern: Decouple graph data type
Idea: Mimic maze exploration

DFS (to visit a vertex v)
• **Mark v as visited.** • **Recursively visit all unmarked vertices w adjacent to v.**

Typical applications:

- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

Challenge:

- How to implement?

Design pattern for graph processing

Goal: Systematically search through a graph from graph processing

- **Create a graph object**
- **Pass the graph to a graph processing routine**
- **Query the graph-processing routine**

public class Paths		
	Paths(Graph G, int s)	*find paths in G from source s*
boolean	hasPathTo(int v)	*is there a path from s to v?*
Iterable<Integer>	pathTo(int v)	*path from s to v; null if no such path*

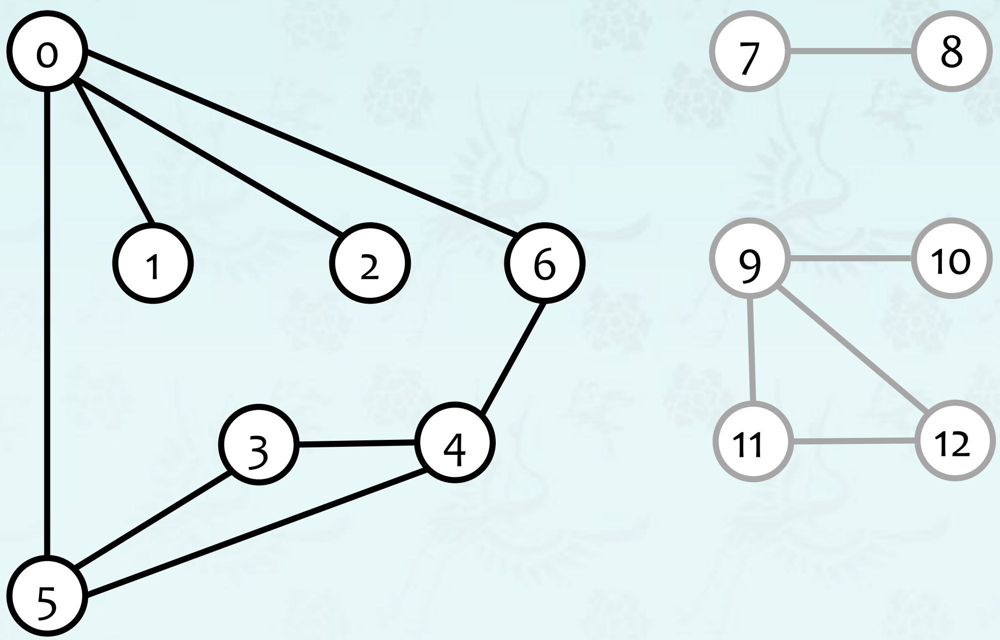
```
Paths paths = new Paths(G, s);
for (int v = 0; v < G.V(); v++)
    if (paths.hasPathTo(v))
        StdOut.println(v);
```

print all vertices connected to s

Build adjacency list

For each edge(v, w) in the list

- Insert front each vertex both (adj[v] , w) and (adj[w], v)
  addEdgeUniDirection(g, v, w);   // add an edge from v to w.



V-E lists

```
myG.txt
13    V
13    E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

Graph g:

Challenge: build adjacency lists?

Adjacency-list graph representation: C implementation 인접리스트

create an empty graph with V vertices

```
pGraph  newGraph(int V) {
   pGraph  g = (pGraph) malloc(sizeof(Graph));
   assert(g != NULL) ;
   g->V = V;
   g->E = 0;

   // create an array of adjacency list. size of array will be V
   g->adj = (pGNode)malloc(V * sizeof(GNode));
   assert(g->adj != NULL);

   // initialize each adjacency list as empty by making head as NULL;

   for (int  i = 0; i < V; i++)
      g->adj[i].next = NULL;
      g->adj[i].item = i
   return g;
}
```

```
typedef struct Graph *pGraph;
typedef struct Graph {
  int          V;          // num of vertices in G
  int          E;          // num of edges G
  pGNode       adj;        // an array of adj lists
} Graph;
```

adjacency list (using an array)

adjacency list set head node NULL

unused; but may store the size of degree.

Adjacency-list graph representation: C implementation

인접리스트

```
// add an edge to an undirected graph

void addEdgeUniDirection(pGraph g, int v, int w) {
   // add an edge from v to w.
   // A new node is added to the adjacency list of v.
   // The node is added at the beginning

   pGNode node = newGNode(w);
   node->next = g->adj[v].next;
   g->adj[v].next = node;
}

// add an edge to an undirected graph

void addEdge(pGraph g, int v, int w) {
   addEdgeUniDirection(g, v, w);        // add an edge from v to w.
   addEdgeUniDirection(g, w, v);        // if graph is undirected, add both
}
```

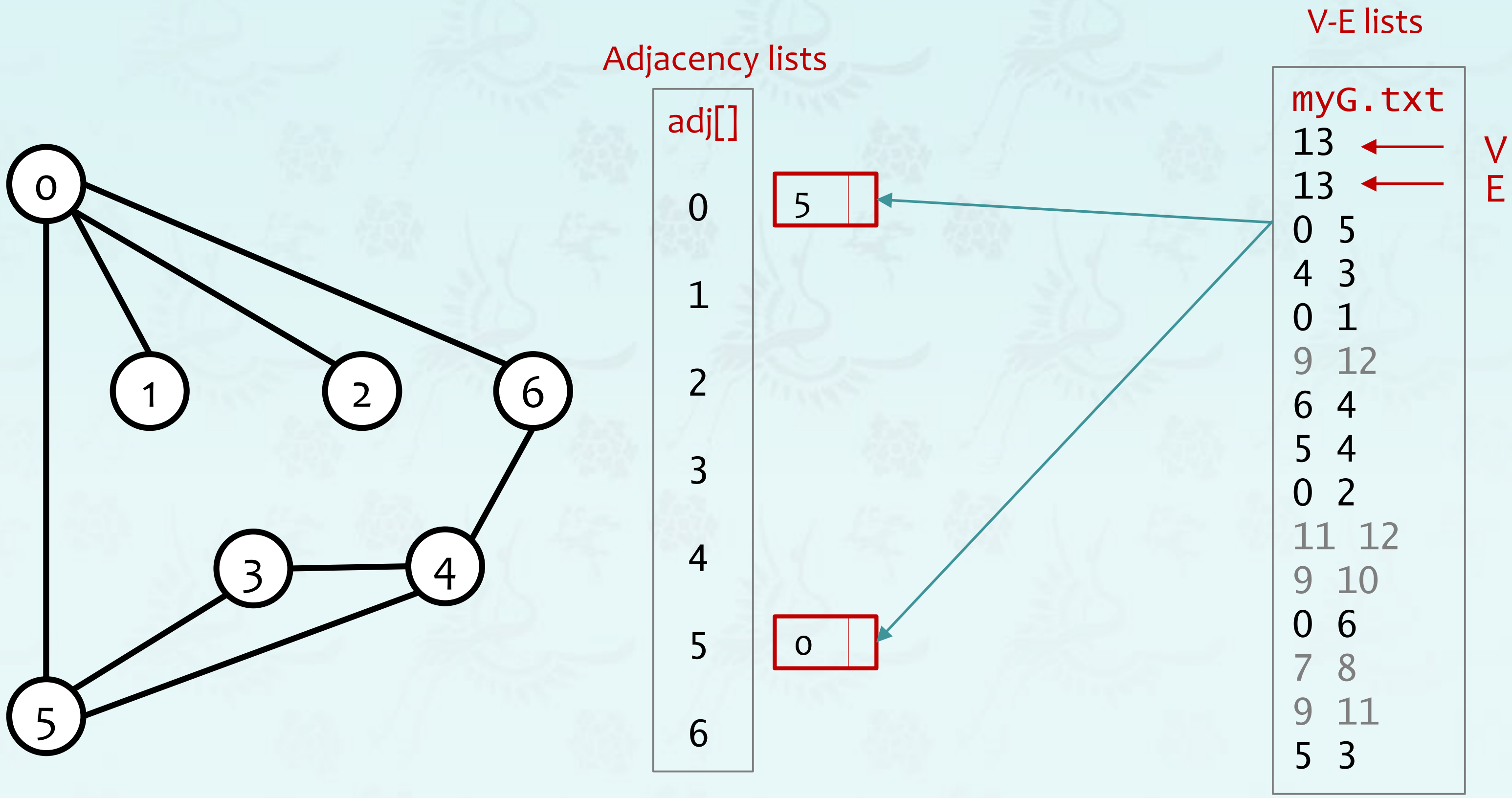
instantiate a node w insert it **at the front** of adjacency list[v]

add an edge for undirected graph

Build adjacency list

For each edge(v, w) in the list

- Insert front each vertex both (adj[v] , w) and (adj[w], v)
  addEdgeUniDirection(g, v, w);   // add an edge from v to w.

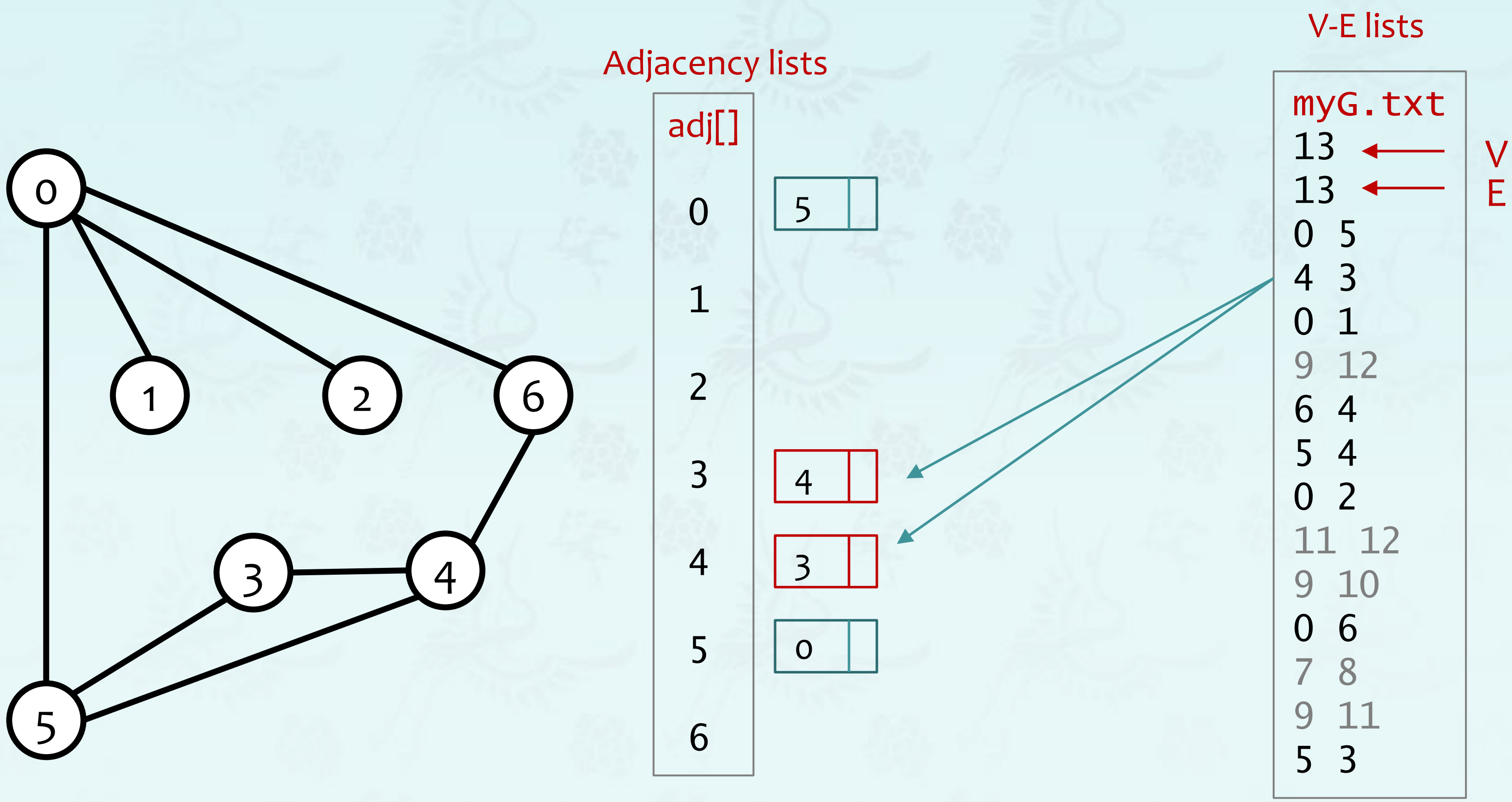


Graph g

Build adjacency list

For each edge(v, w) in the list

- Insert front each vertex both (adj[v] , w) and (adj[w], v)
  addEdgeUniDirection(g, v, w); // add an edge from v to w.

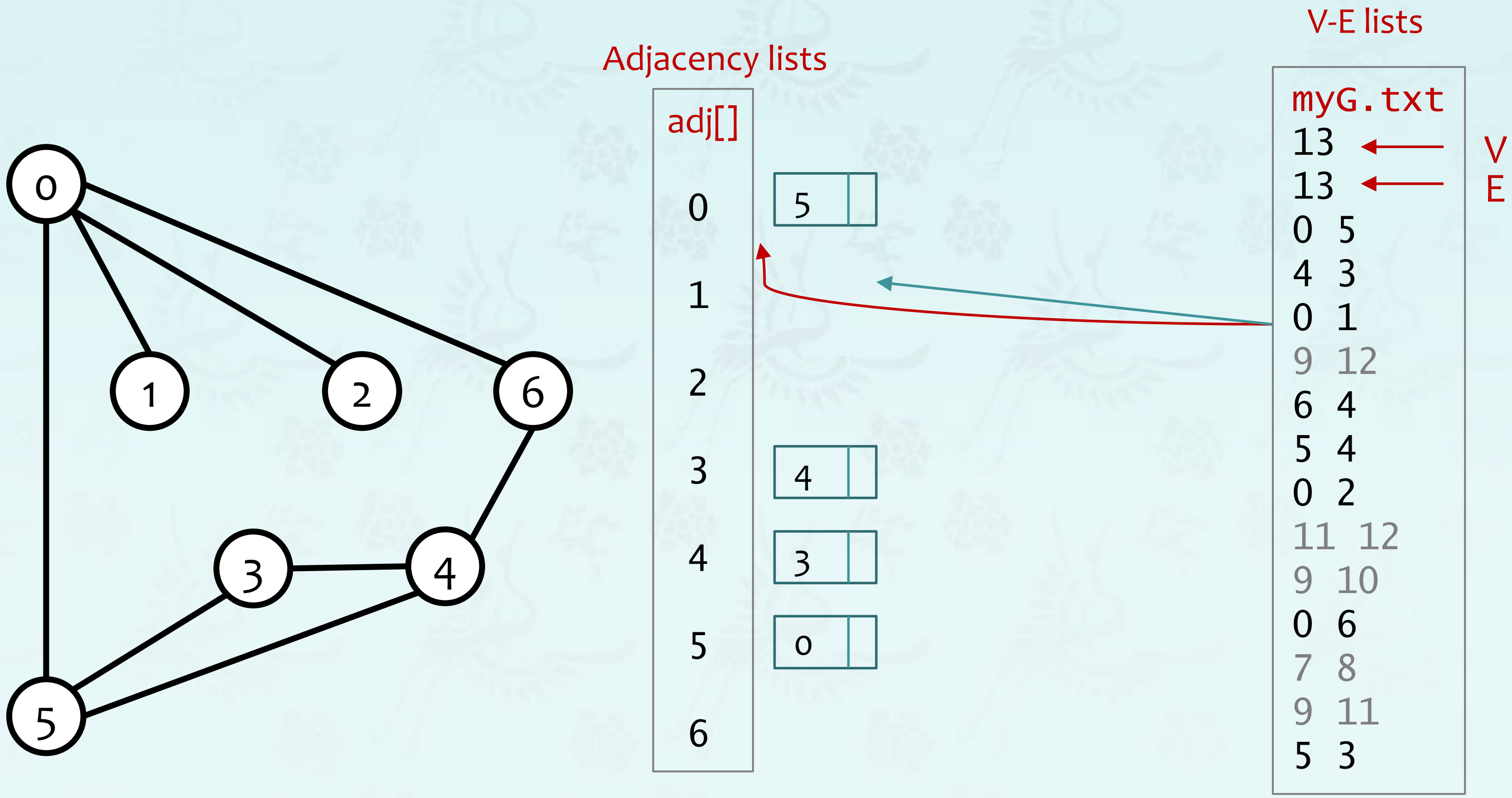


Graph g

Build adjacency list

For each edge(v, w) in the list

- Insert front each vertex both (adj[v] , w) and (adj[w], v)
  addEdgeUniDirection(g, v, w);   // add an edge from v to w.

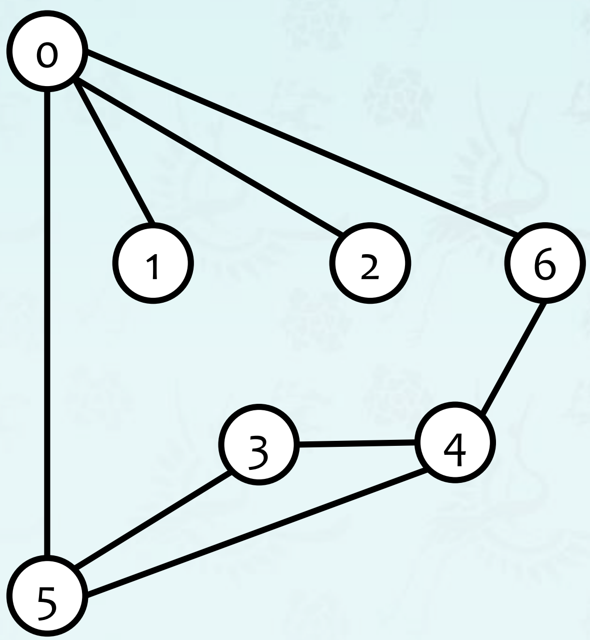


Graph g

Build adjacency list

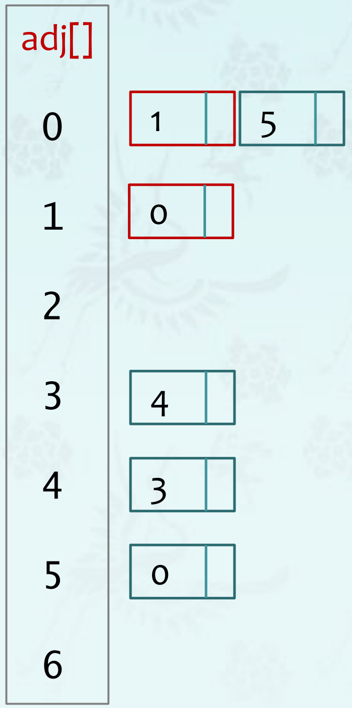
For each edge(v, w) in the list

- Insert front each vertex both (adj[v] , w) and (adj[w], v)
  addEdgeUniDirection(g, v, w);   // add an edge from v to w.



Graph g

Adjacency lists



V-E lists

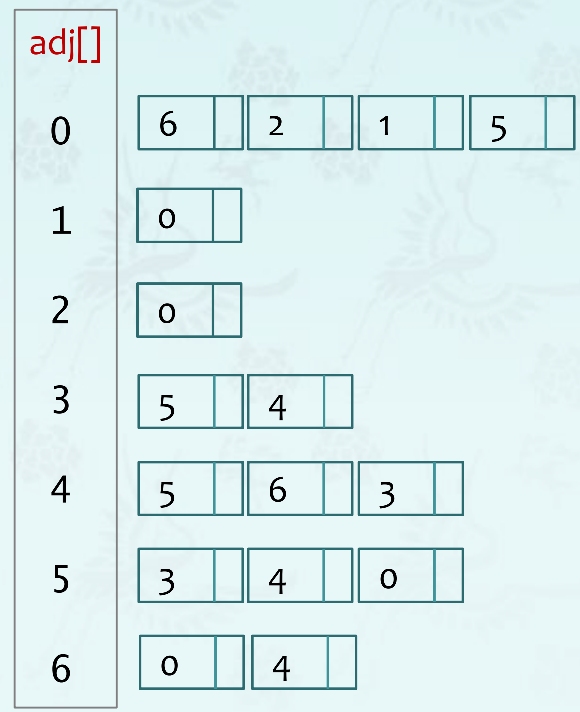
```
myG.txt
13    ← V
13    ← E
0  5
4  3
0  1
9  12
6  4
5  4
0  2
11  12
9  10
0  6
7  8
9  11
5  3
```

Build adjacency list

For each edge(v, w) in the list

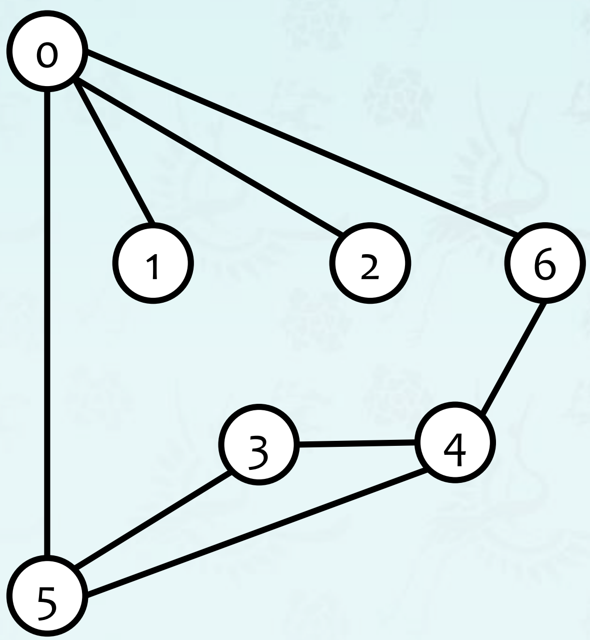
- Insert front each vertex both (adj[v] , w) and (adj[w], v)
  addEdgeUniDirection(g, v, w); // add an edge from v to w.

Adjacency lists



V-E lists

```
myG.txt
13   ← V
13   ← E
0  5
4  3
0  1
9  12
6  4
5  4
0  2
11  12
9  10
0  6
7  8
9  11
5  3
```



Graph g

Depth-first search demo

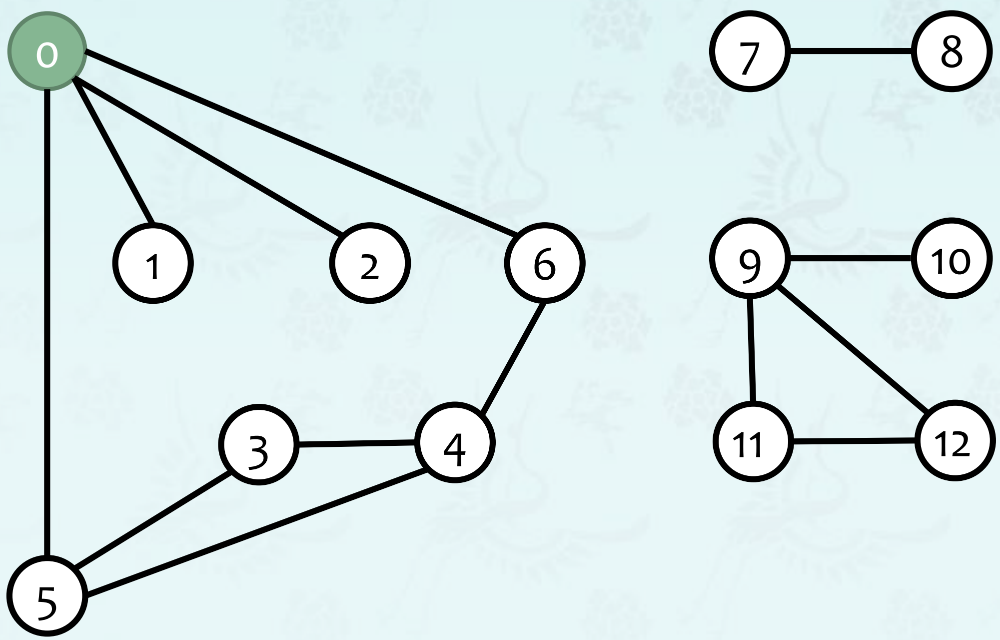
To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



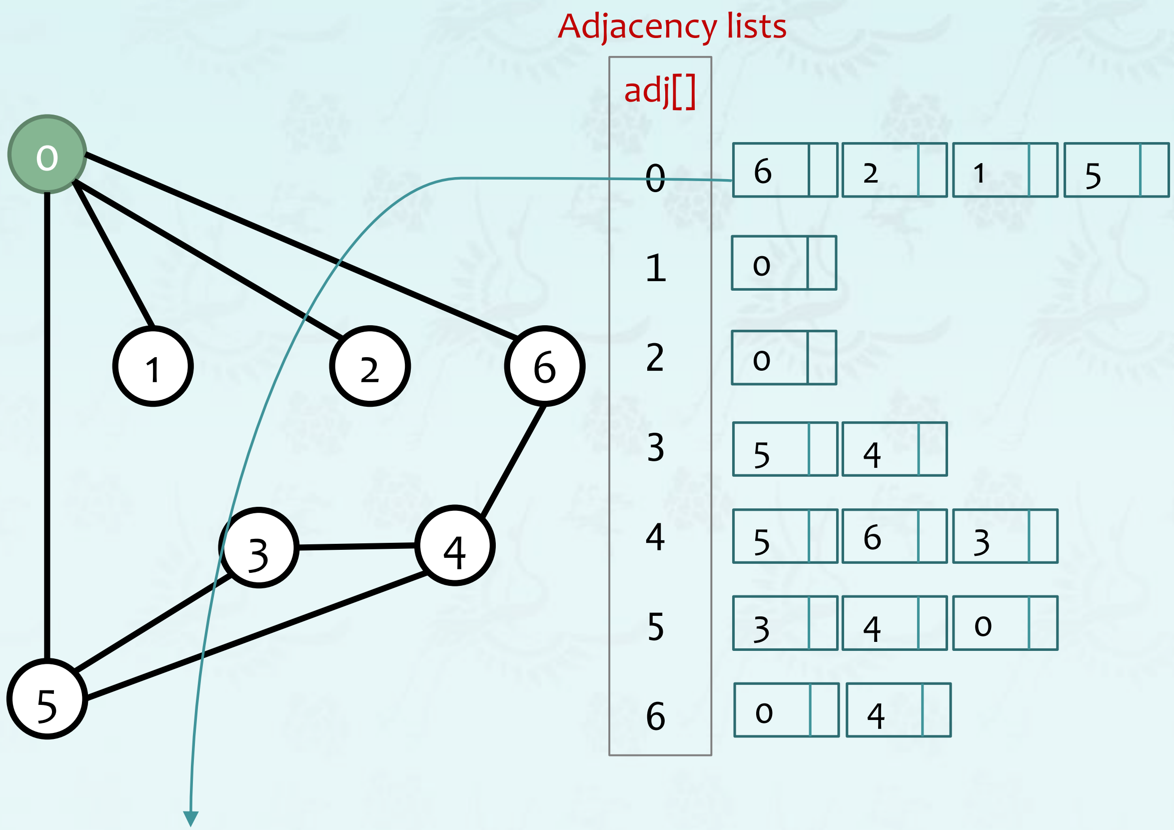
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: **Which one first?**

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



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

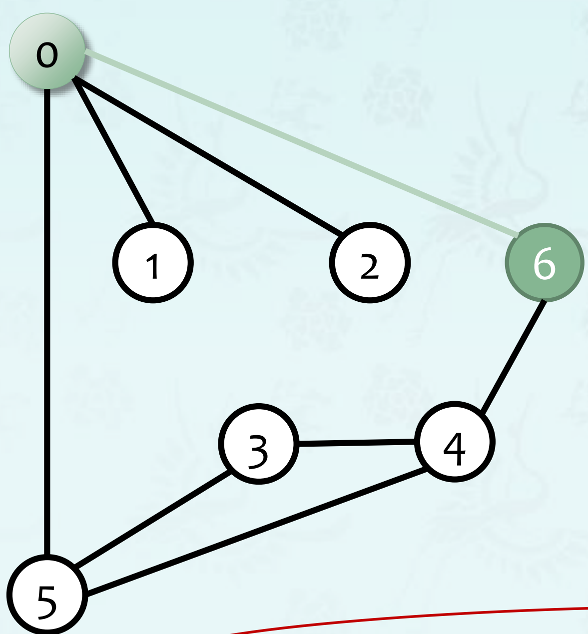
visit 0: **check 6**, **check 2, check 1, and check 5**

DFS 0

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	5	4		
4	5	6	3	
5	3	4	0	
6	~~0~~	4		

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

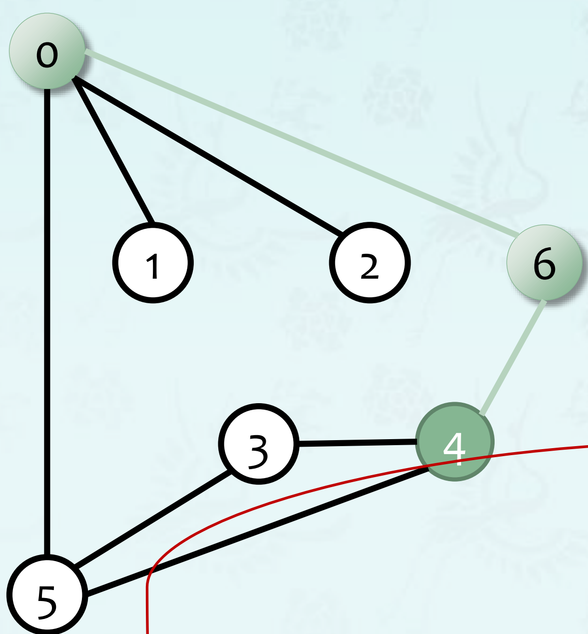
visit 6: **~~check 0~~**, check 4

DFS 0 6

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	5	4		
4	5	6	3	
5	3	4	0	
6	~~0~~	4		

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

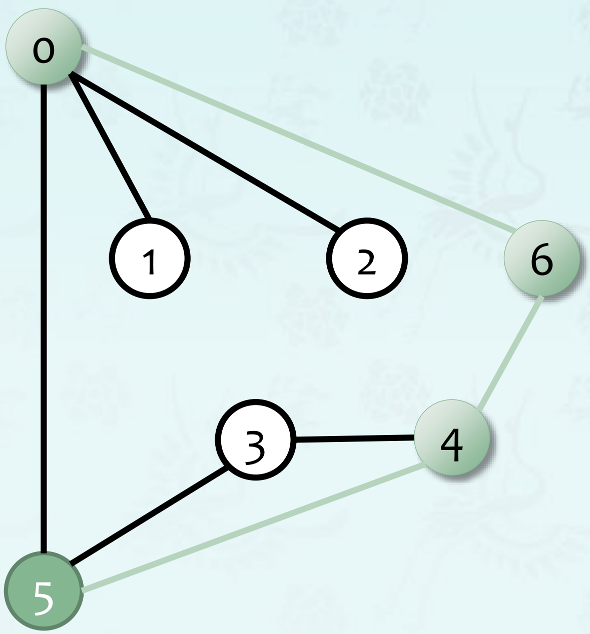
visit 4: **check 5**, **check 6, check 3**

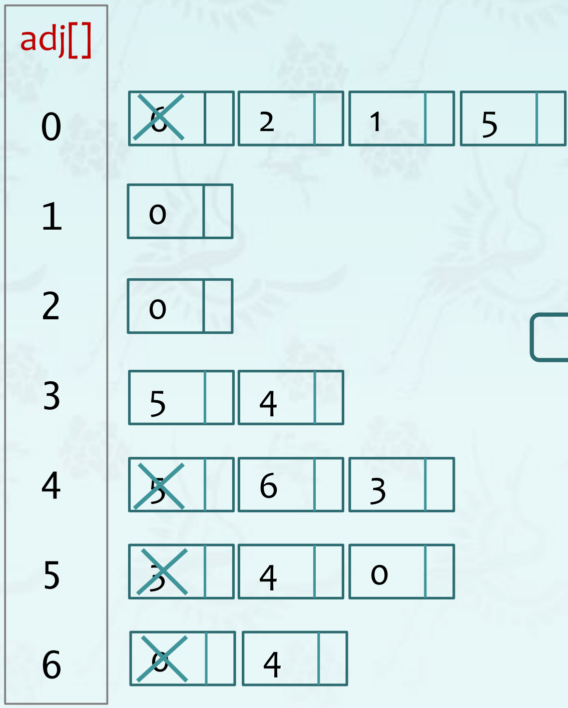
DFS 0 6 4

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.





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

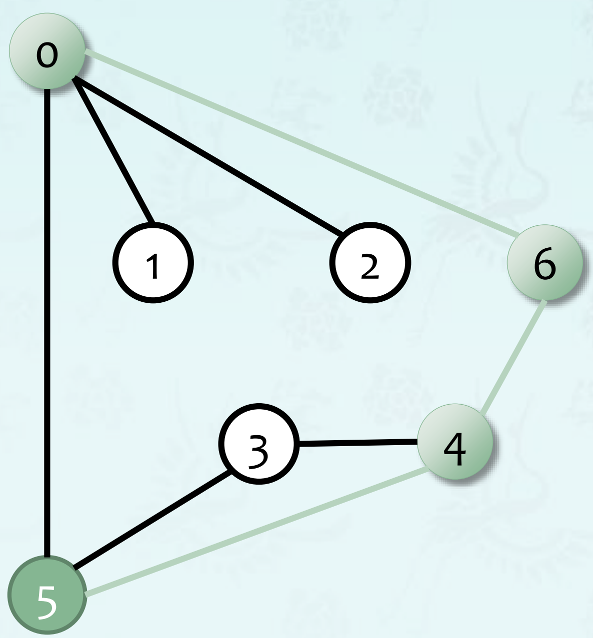
visit 5: **check 3**, **check 4, check 0**

DFS 0 6 4 5

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



adj[]				
0	6	2	1	5
1	0			
2	0			
3	5	4		
4	5	6	3	
5	3	4	0	
6	0	4		

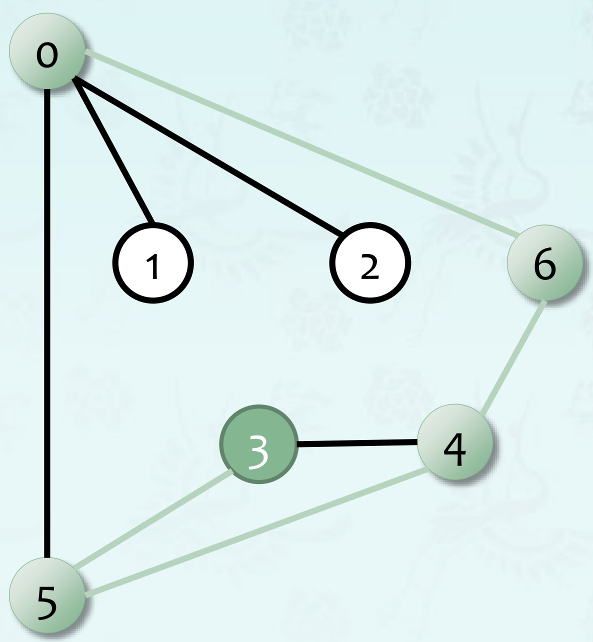
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	F	–
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 5: **check 3**, **check 4**, **check 0**

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



adj[]				
0	6	2	1	5
1	0			
2	0			
3	5	4		
4	5	6	3	
5	3	4	0	
6	0	4		

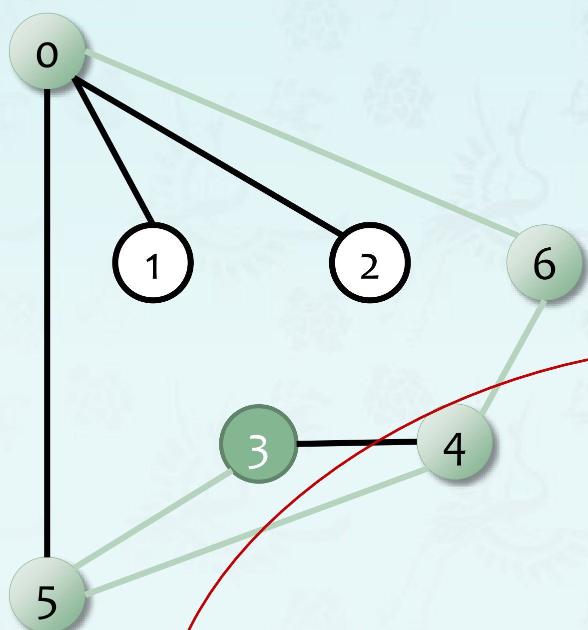
v	marked[]	parent[v]
0	T	-
1	F	-
2	F	-
3	T	5
4	T	6
5	T	4
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

visit 5: **check 3**, **check 4**, **check 0**

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	6	2	1	5
1	0			
2	0			
3	5	4		
4	5	6	3	
5	3	4	0	
6	0	4		

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

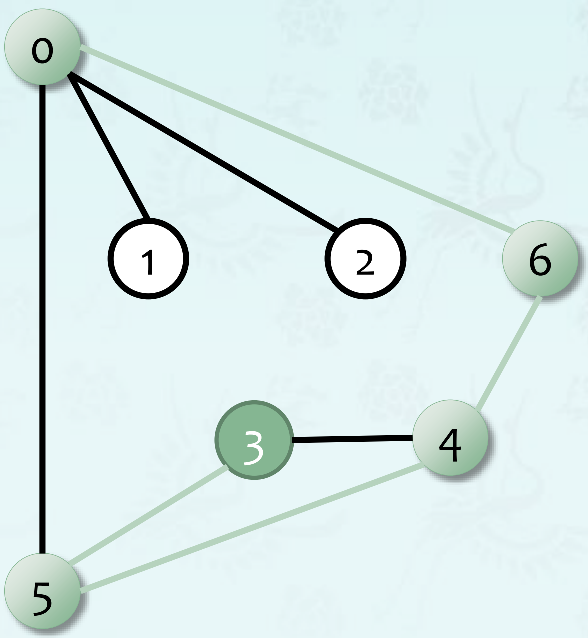
visit 3: **check 5**, **check 4**

DFS 0 6 4 5 3

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	4		
4	~~5~~	6	3	
5	~~3~~	4	0	
6	~~0~~	4		

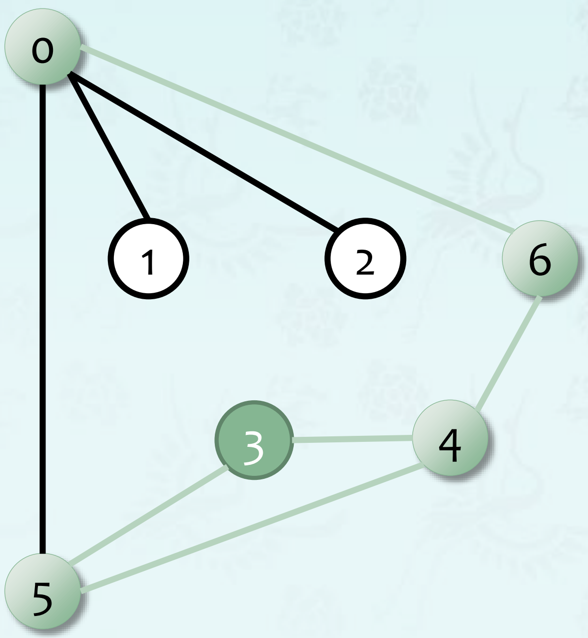
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 3: ~~**check 5**~~, **check 4**

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	4		
4	~~5~~	6	3	
5	~~3~~	4	0	
6	~~0~~	4		

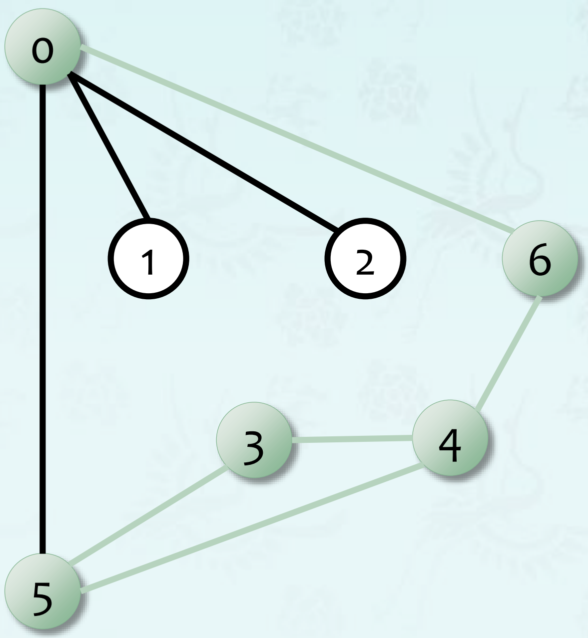
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 3: check 5, ~~check 4~~

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	4	0	
6	~~0~~	4		

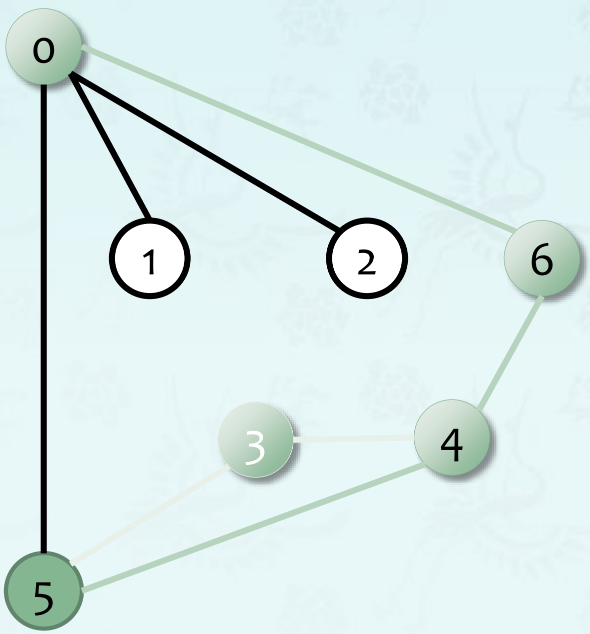
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 3: check 5, ~~check 4~~

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	4	0	
6	~~0~~	4		

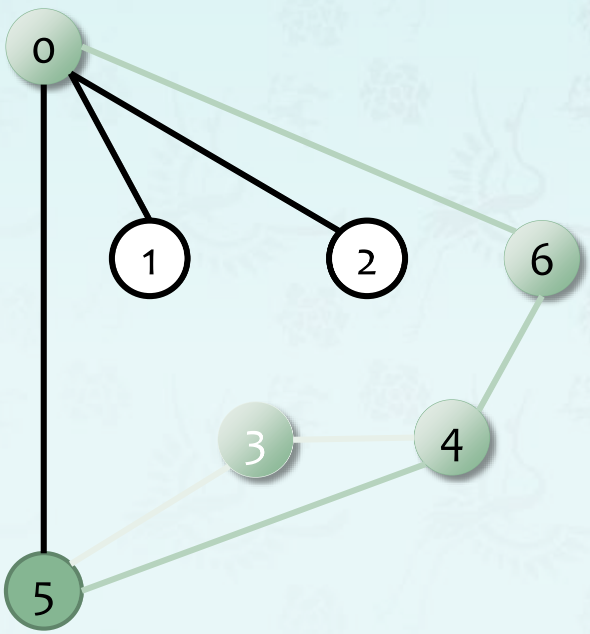
v	marked[]	parent[v]
0	T	-
1	F	-
2	F	-
$\boxed{3}$	$\boxed{T}$	$\boxed{5}$
4	T	6
5	T	4
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

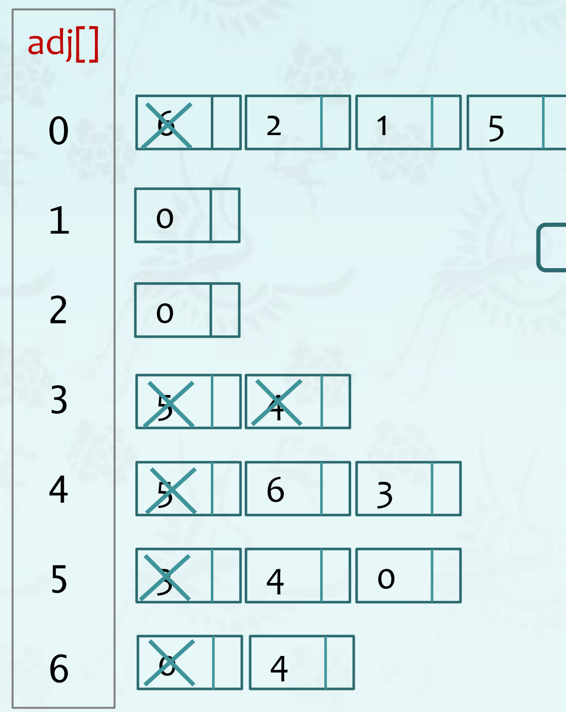
3 done: What's the next?

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.





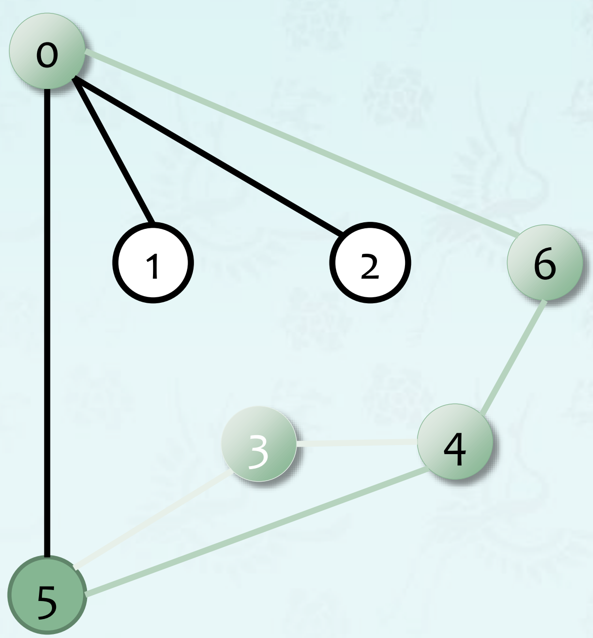
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

3 done: What's the next? **Backtrack!**

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	4	0	
6	~~0~~	4		

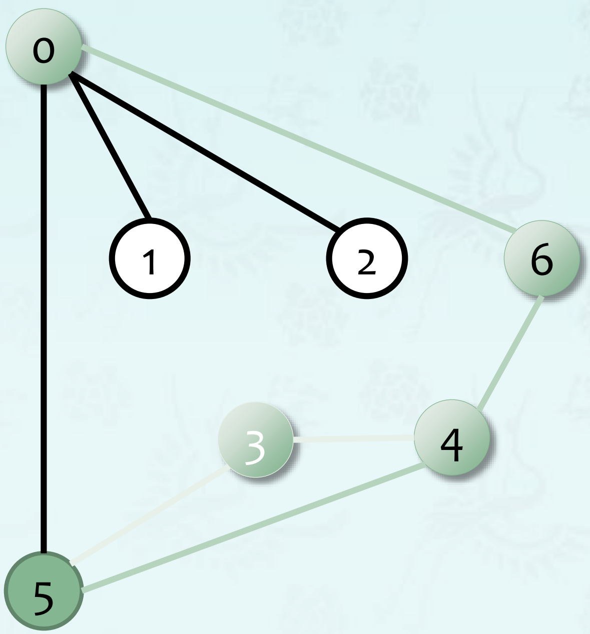
v	marked[]	parent[v]
0	T	-
1	F	-
2	F	-
3	T	5
4	T	6
5	T	4
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

3 done: What's the next? **Backtrack!**
How to?

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	4	0	
6	~~0~~	4		

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

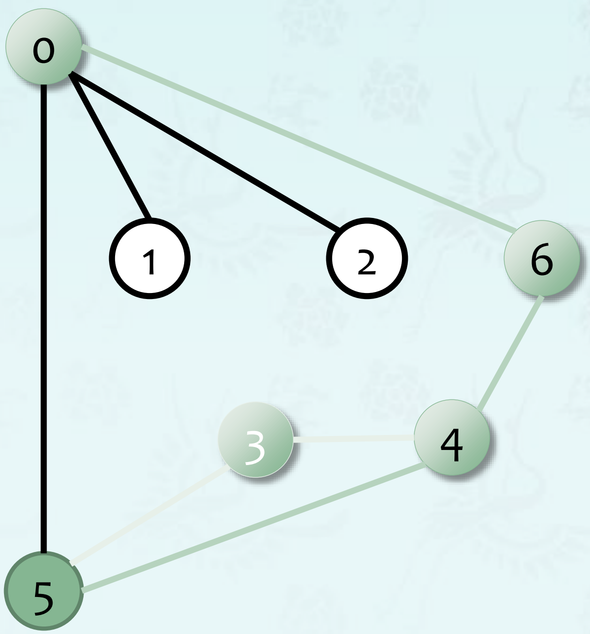
3 done: What's the next? **Backtrack!**

How to? **Use parent[v]** parent[3] = 5

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	4	0	
6	~~0~~	4		

v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
$\boxed{5}$	$\boxed{T}$	$\boxed{4}$
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

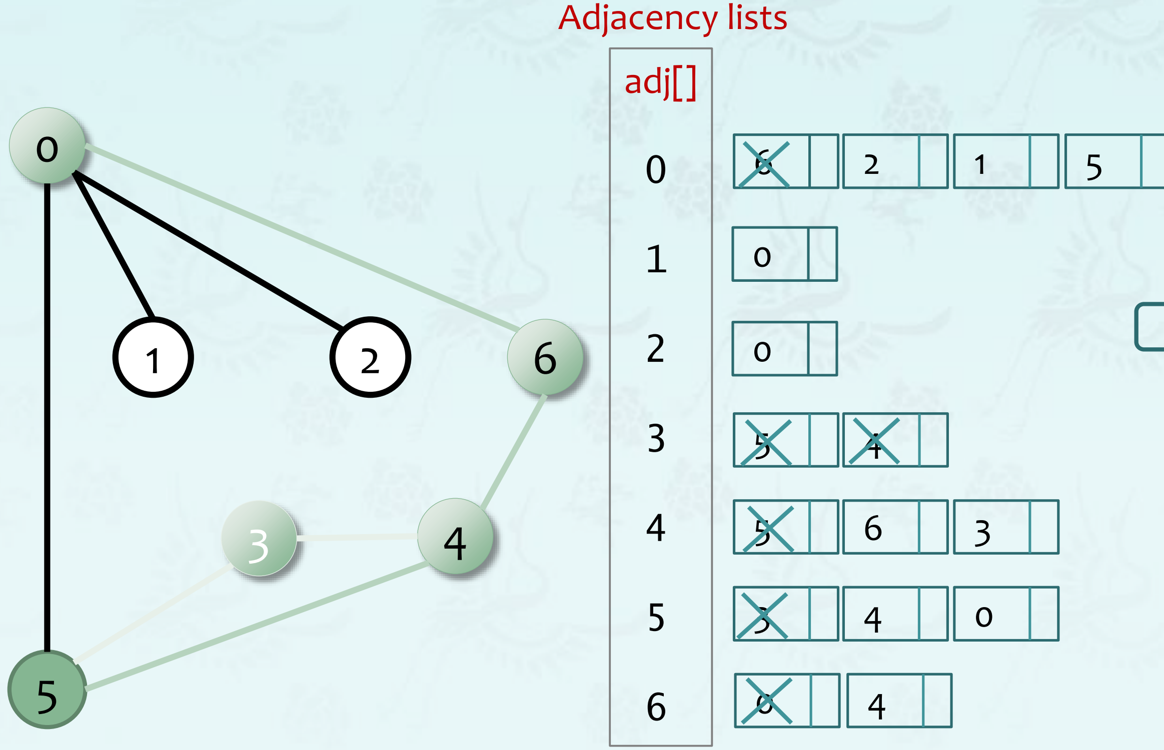
done

visit 5: check 3, **check 4**, **check 0**

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



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

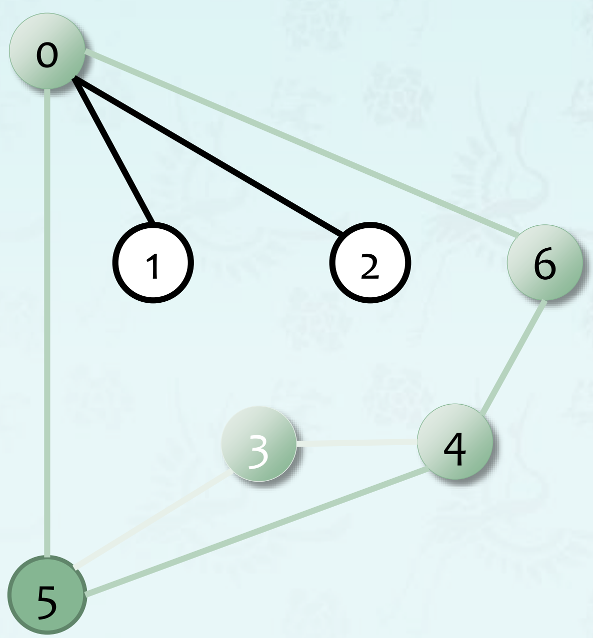
done

visit 5: check 3, **check 4**, check 0

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	~~4~~	0	
6	~~0~~	4		

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

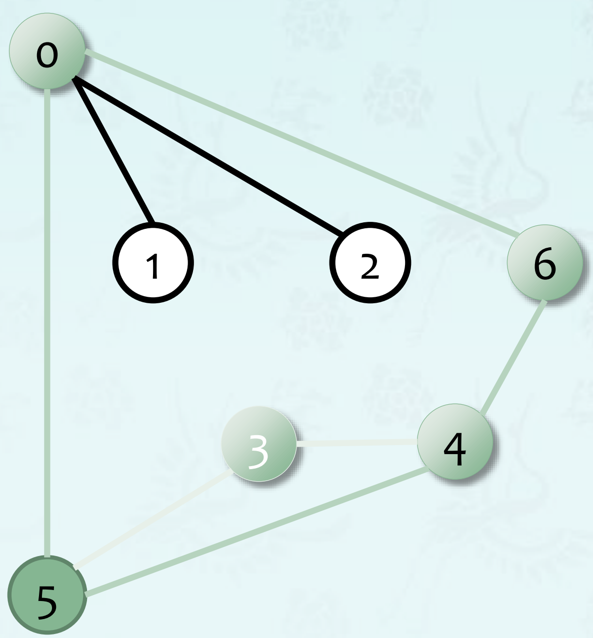
done

visit 5: check 3, ~~check 4~~, **check 0**

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



adj[]

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

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

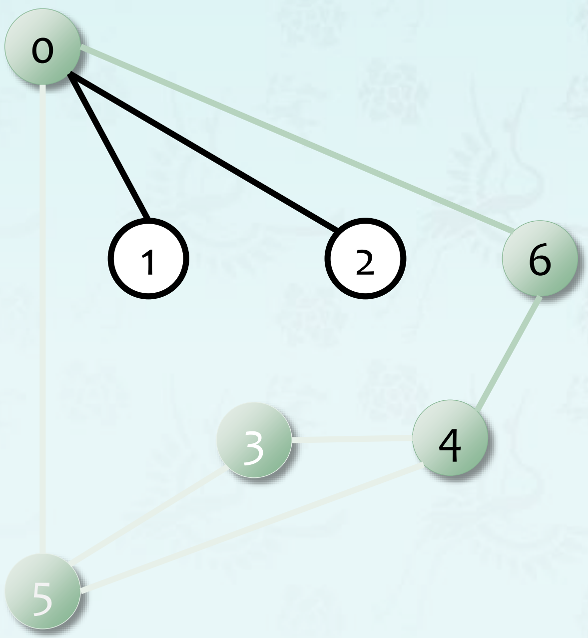
done

visit 5: check 3, check 4, **check 0**

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	4		

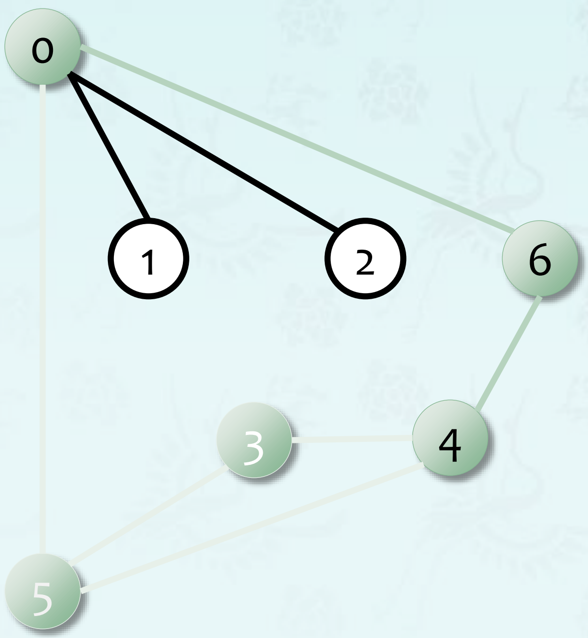
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

5 done

What's the next? **Backtrack!**
How to?

Depth-first search demo

To visit a vertex v:
- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	6	3	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	4		

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

5 done

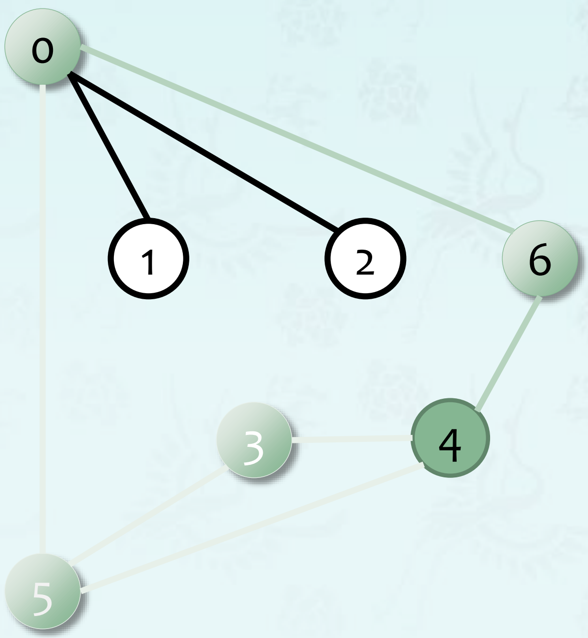
What's the next? **Backtrack!**
How to? **Use parent[v]**

edgetTo[5] = 4

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



adj[]				
0	6	2	1	5
1	0			
2	0			
3	5	4		
4	5	6	3	
5	3	4	0	
6	0	4		

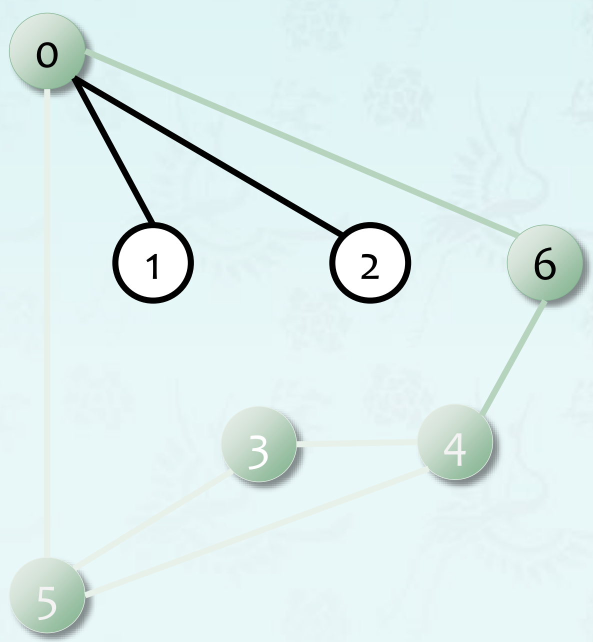
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 4: check 5, **check 6**, **check 3**

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	~~6~~	~~3~~	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	4		

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

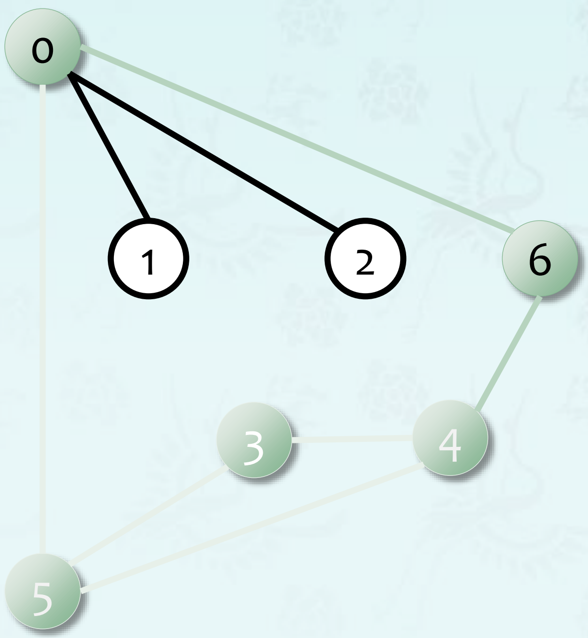
visit 4: check 5, check 6, **check 3**

4 done

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	~~6~~	~~3~~	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	4		

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

visit 4: check 5, check 6, **check 3**

4 done

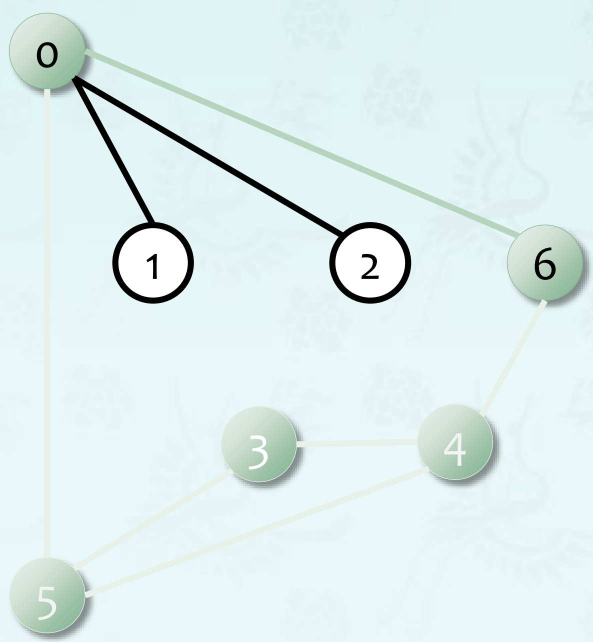
Backtrack!

parent[4] = 6

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	~~6~~	~~3~~	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	4		

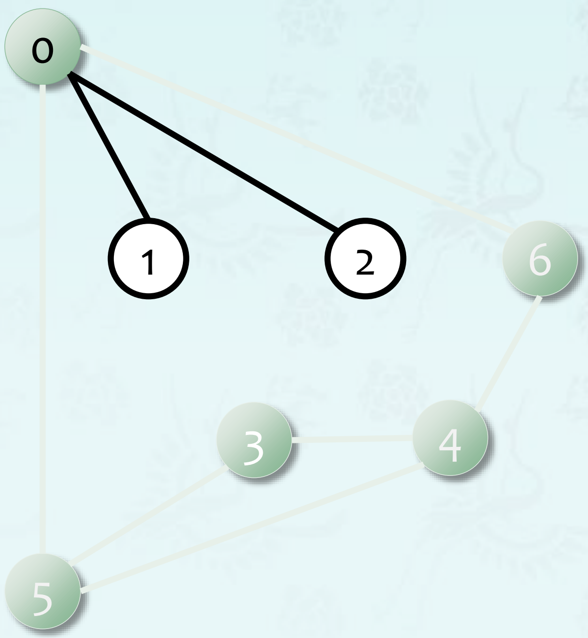
v	marked[]	parent[v]
0	T	–
1	F	–
2	F	–
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 6: check 0, check 4

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]	
0	~~6~~ 2 1 5
1	0
2	0
3	~~5~~ ~~4~~
4	~~5~~ ~~6~~ ~~3~~
5	~~3~~ ~~4~~ ~~0~~
6	~~0~~ ~~4~~

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

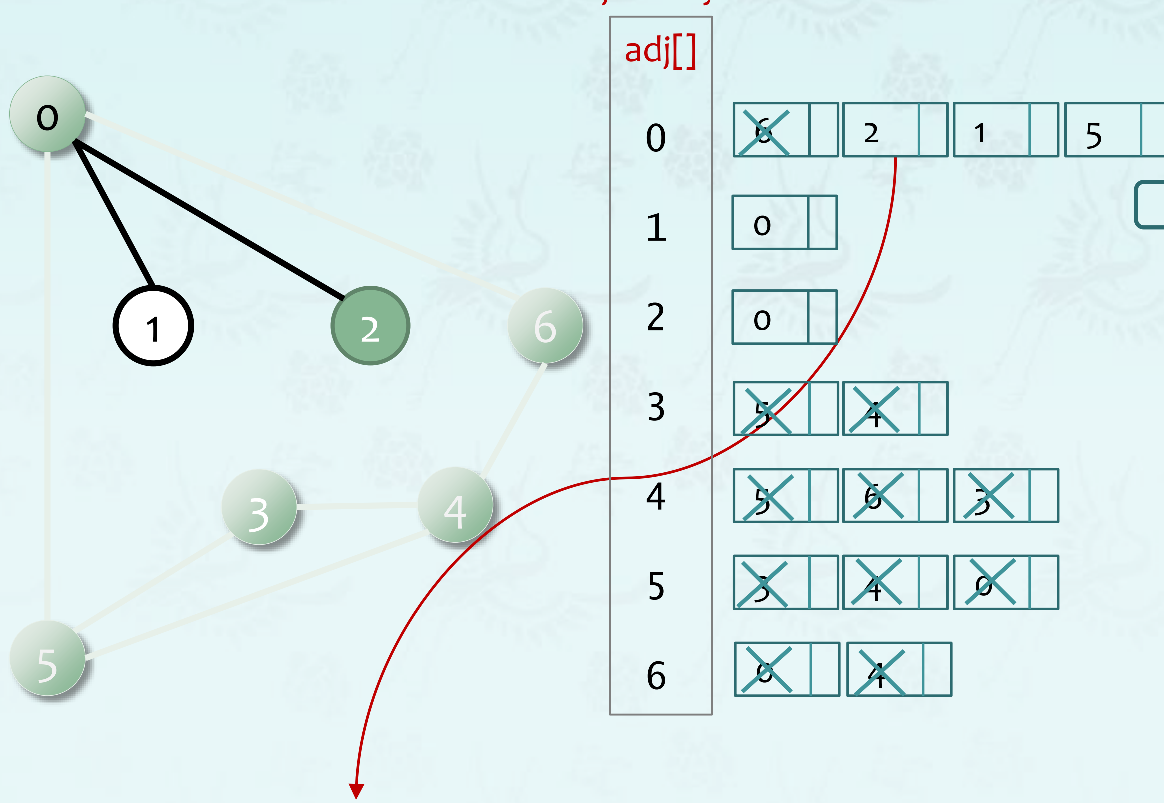
visit 6: check 0, check 4 **done 6** **Backtrack!** **parent[6] = 0**

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

Adjacency lists



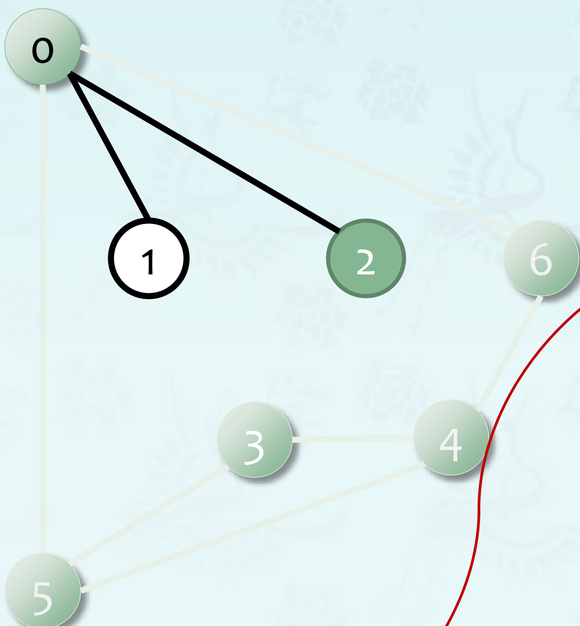
v	marked[]	parent[v]
0	T	–
1	F	–
$\boxed{2}$	$\boxed{T}$	$\boxed{0}$
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: check 6, **check 2**, **check 1, and check 5**

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]				
0	~~6~~	2	1	5
1	0			
2	0			
3	~~5~~	~~4~~		
4	~~5~~	~~6~~	~~3~~	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	~~4~~		

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

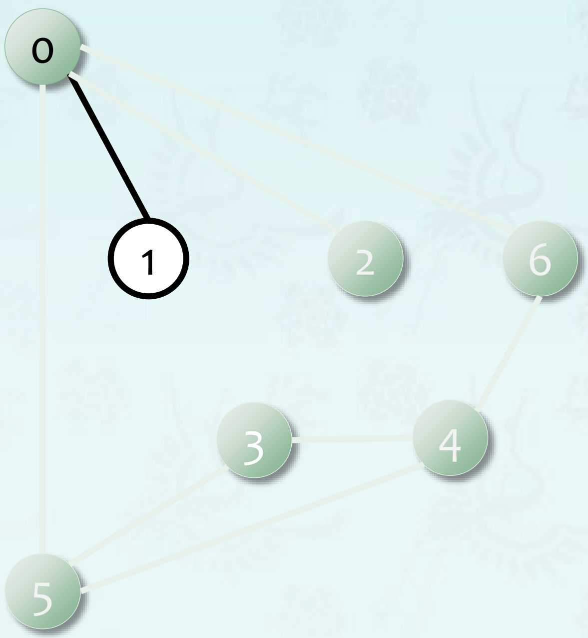
visit 2: check 0

DFS: 0 6 4 5 3 2

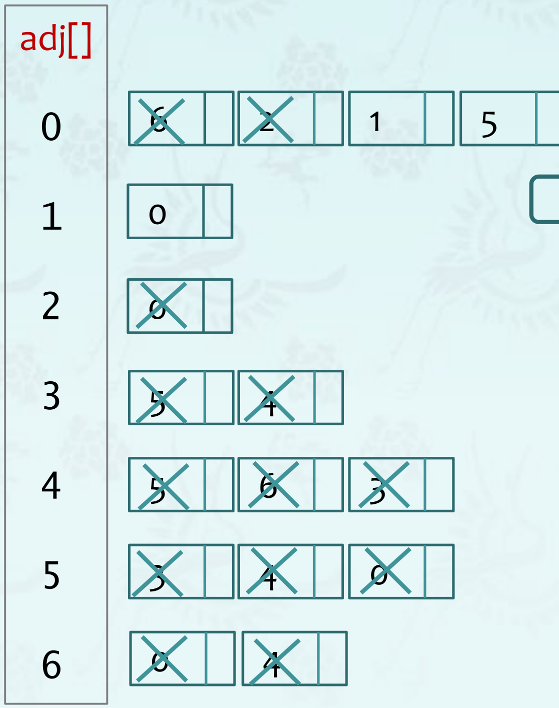
Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists



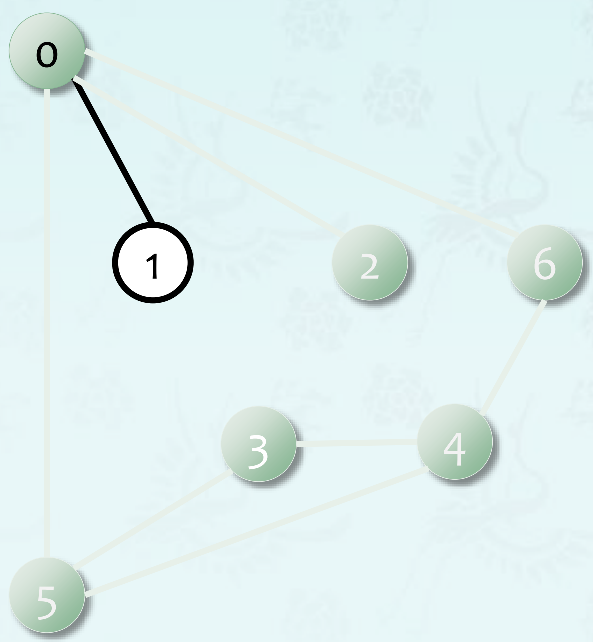
v	marked[]	parent[v]
0	T	–
1	F	–
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 2: check 0 **2 done** **Backtrack!** **parent[2] = 0**

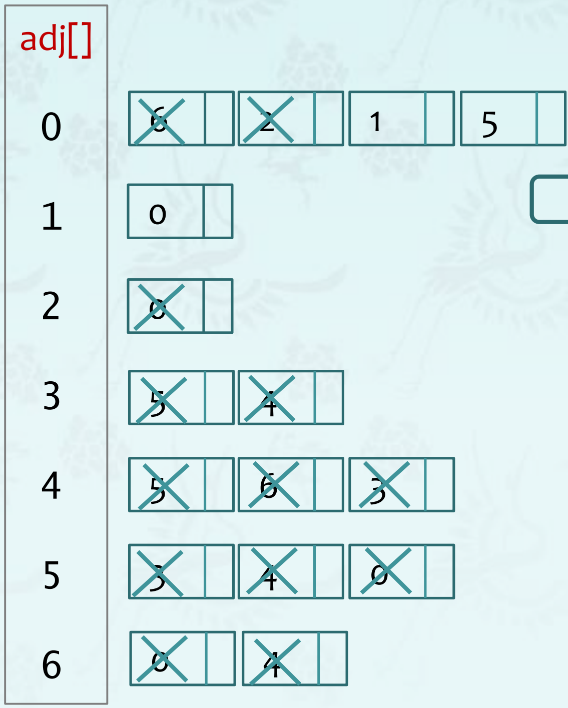
Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists



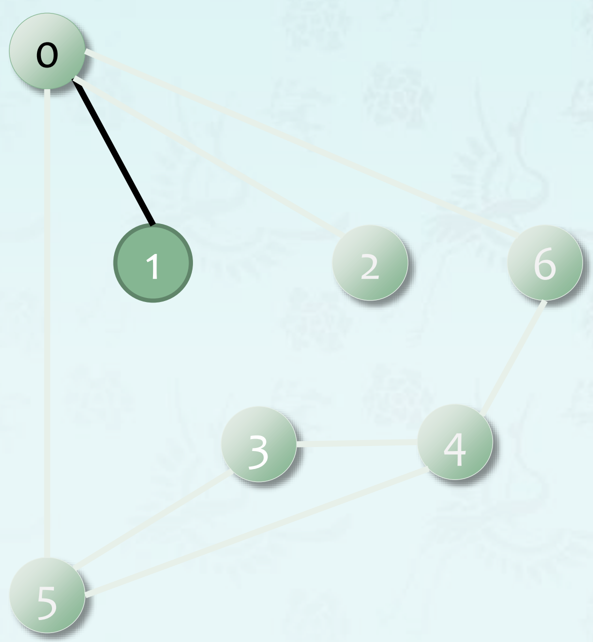
v	marked[]	parent[v]
0	T	–
1	F	–
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: check 6, check 2, **check 1**, **and check 5**

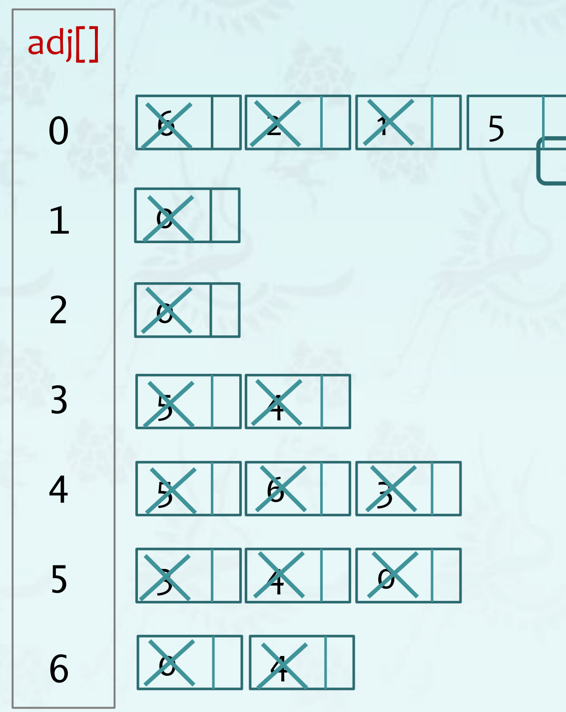
Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists



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

visit 1: check 0

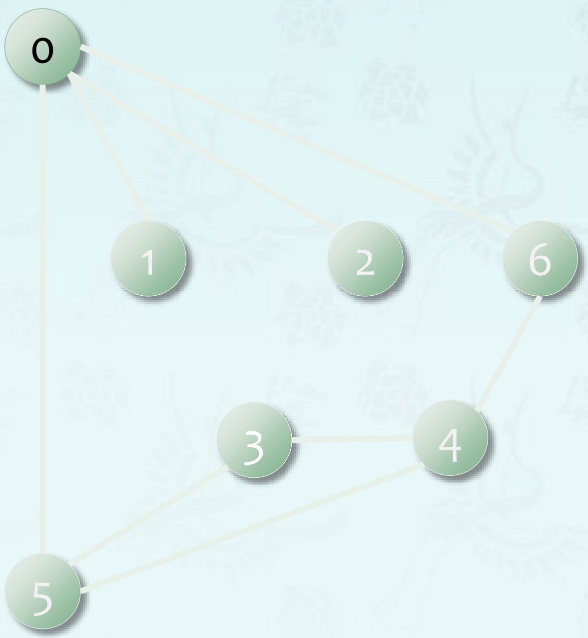
1 done **Backtrack!** **parent[1] = 0**

DFS: 0 6 4 5 3 2 1

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]

0	~~6~~	~~2~~	~~1~~	5
1	~~0~~			
2	~~0~~			
3	~~5~~	~~4~~		
4	~~5~~	~~6~~	~~3~~	
5	~~3~~	~~4~~	~~0~~	
6	~~0~~	~~4~~		

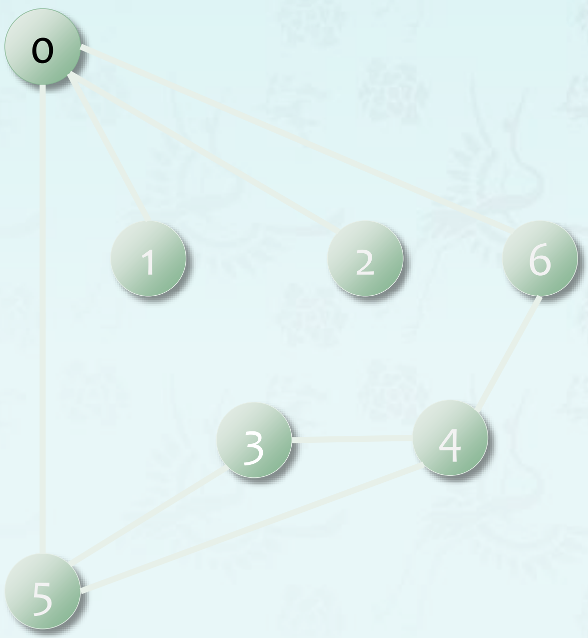
v	marked[]	parent[v]
0	T	–
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

visit 0: check 6, check 2, check 1, and **check 5**

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Adjacency lists

adj[]

0: ~~6~~ ~~2~~ ~~1~~ ~~5~~
1: ~~0~~
2: ~~0~~
3: ~~5~~ ~~4~~
4: ~~5~~ ~~6~~ ~~3~~
5: ~~3~~ ~~4~~ ~~0~~
6: ~~0~~ ~~4~~

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

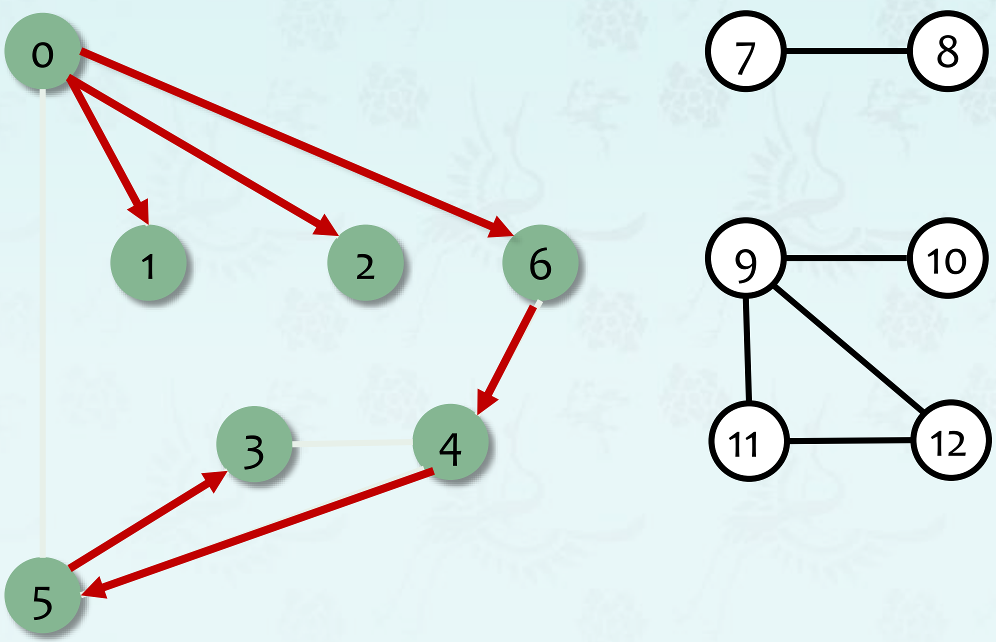
visit 0: check 6, check 2, check 1, and check 5

0 done

Depth-first search demo

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



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

DFS Output: DFS: 0 6 4 5 3 2 1

- found vertices reachable from 0
- build a data structure **parent[v]**

Depth-first search

Goal: Find all vertices connected to s (and a corresponding path).
Idea: Mimic maze exploration

Algorithm:

- Use recursion (ball of string).
- Mark each visited vertex (and keep track of edge taken to visit it).
- Return (retrace steps) when no unvisited options.

Data Structures:

- `Boolean[] marked` to mark visited vetices.
- `int[] parent` to keep tree of paths.
  (`parent[w] == v`) means that edge v-w taken to visit w for first time

Depth-first search Implementation in C

```
// DFS - find vertices connected to v
void depthFirstSearch(pGraph g, int v){
  short *marked = (short *)malloc(V(g) * sizeof(short)); assert(marked!=NULL);
  int   *parent = (int   *)malloc(V(g) * sizeof(int  )); assert(marked!=NULL);

  for (int i = 0; i < V(g); i++) {
        marked[i] = false;
        parent[i] = -1;
  }
  dfs(g, v, marked, parent);        // printf("Depth First Search: ");
  free(marked);
  free(parent);                     // we may keep this info pGraph g.
}
```

this function does the job recursively

Depth-first search Implementation in C

```
// Recursive DFS does the work


void dfs(pGraph g, int v, short *visited, int *parent) {
  visited[v] = true;
  printf("%d ", v);
  loop through from the current node v to the rest)
    if this node w is not visited
        dfs ( with w )
        update parent with w & v
}
```

visiting node currently

where it reached from

Depth-first search properties

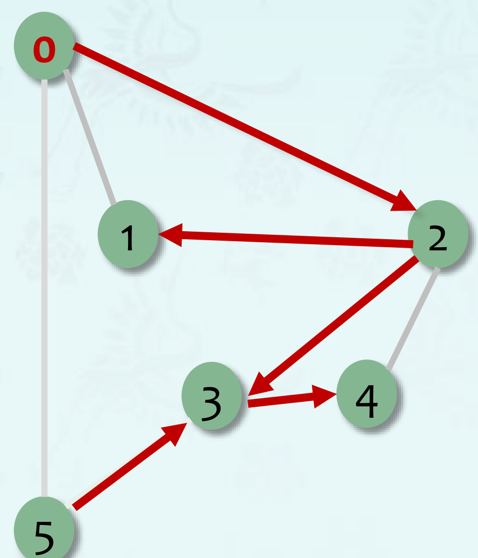
Proposition: After DFS, can find vertices connected to s in constant time and can find a path to s (if one exists) in time proportional to its length.

Pf: parent[] is parent-link representation of a tree rooted at s.

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

public Iterable<Integer> pathTo(int v){
  if (!hasPathTo(v)) return null;

  Stack<Integer> path = new Stack<Integer>();
  for (int x = v; x != s; x = parent[x])
    path.push(x);
  path.push(s);
  return path;
}
```



	parent[]
0	
1	2
2	0
3	2
4	3
5	3

a wrong edge?

ECE20010 Data Structures

Chapter 6

- *Introduction*
- *Graph API*
- *Elementary Graph Operations*
  - ***DFS: Depth first search***
  - *BFS: Breadth first search*
  - *CC: Connected Components*

Major references:
1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

Prof. Youngsup Kim, idebtor@handong.edu, 2014 Data Structures, CSEE Dept., Handong Global University

ECE20010 Data Structures

Chapter 6

- *Introduction*
- *Graph API*
- *Elementary Graph Operations*
  - *DFS: Depth first search*
  - *BFS: Breadth first search*
  - ***CC: Connected Components***

Major references:
1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

Prof. Youngsup Kim, idebtor@handong.edu, 2014 Data Structures, CSEE Dept., Handong Global University

Connectivity queries

Def.: Vertices v and w are connected if there is a path between them.

Goal: Preprocess graph to answer queries of the form "*is v connected to w?*" in constant time.

public class CC		
	`CC(Graph G)`	*find connected components in G*
`boolean`	`connected(int v, int w)`	*are v and w connected?*
`int`	`count()`	*number of connected components*
`int`	`id(int v)`	*component identifier for v*

Union-Find? Not quite.
Depth-first search? Yes …

Connected components

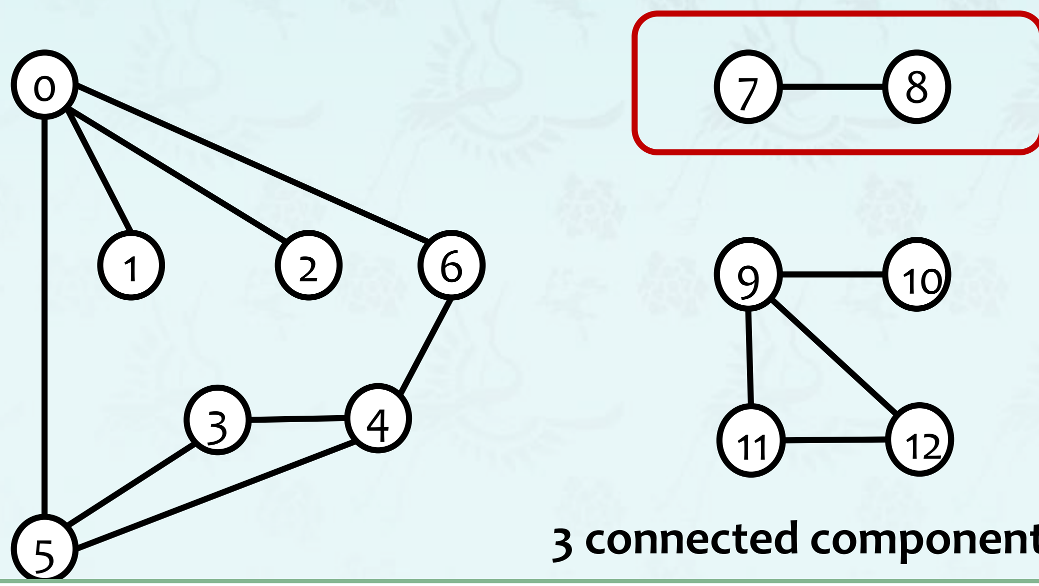
The relation “is connected to” is equivalence relation:

Reflexive: v is connected to v.

Symmetric: if v is connected to w, then w is connected v.

Transitive: if v connected to w and w connected to x, then v connected to x

Def.: A connected component is a maximal set of connected vertices.



3 connected components

v	id[v]
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	1
8	1
9	2
10	2
11	2
12	2

Remark: Given connected components, can answer queries in constant time.

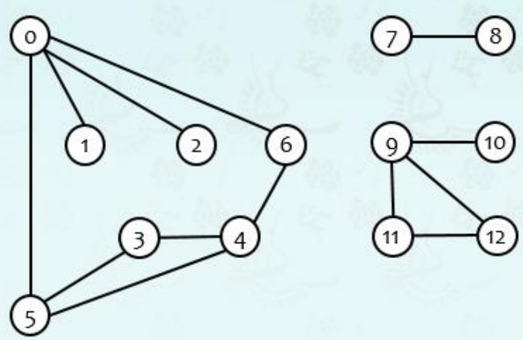
Connected components

Goal: Partition vertices into connected components.

Connected components

Initialize all vertices v as unmarked.

For each unmarked vertex v, run DFS to identify all vertices discovered as part of the same component.

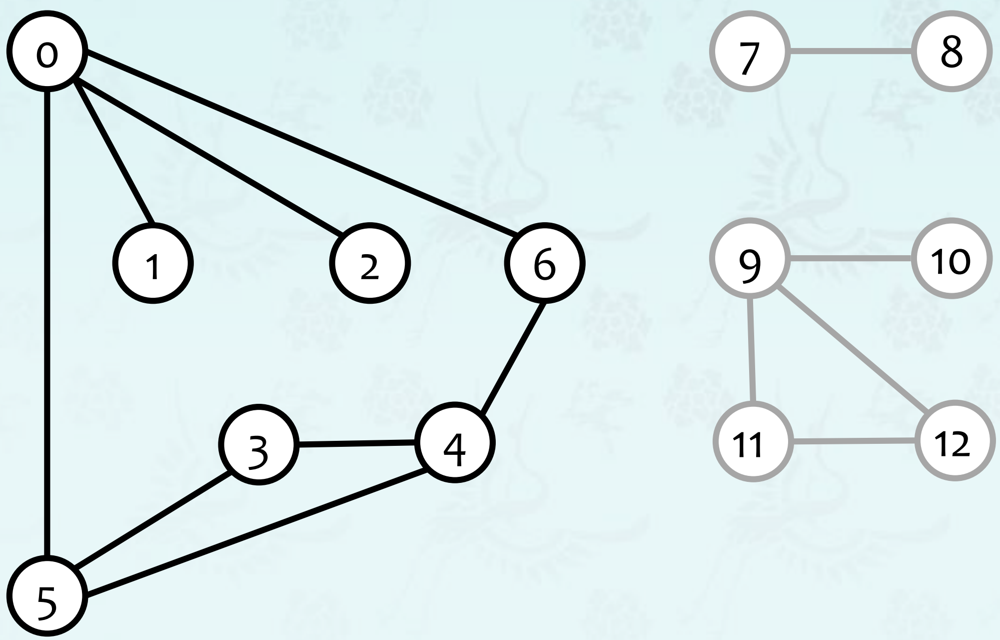



```
myG.txt
13
13
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

Connected components

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



V-E lists →

```
myG.txt
13    ← V
13    ← E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

Graph g:

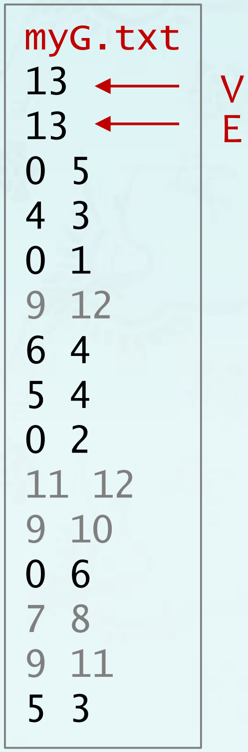
Challenge: build adjacency lists?

Depth-first search demo

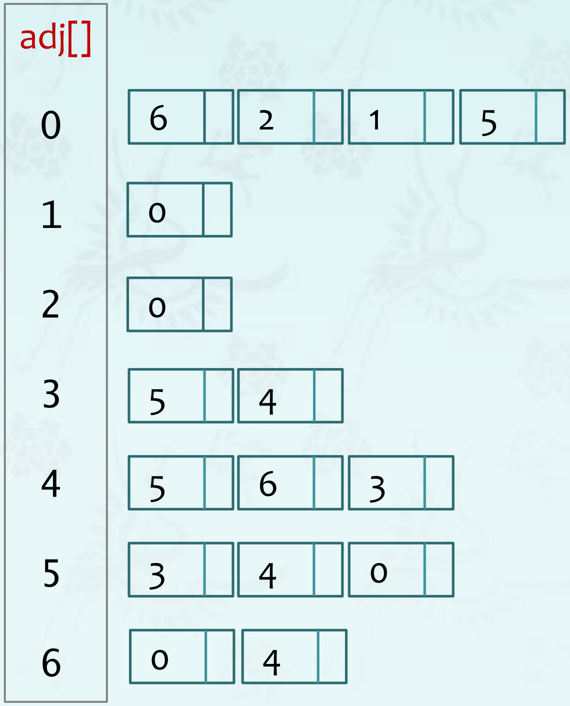
To visit a vertex v:

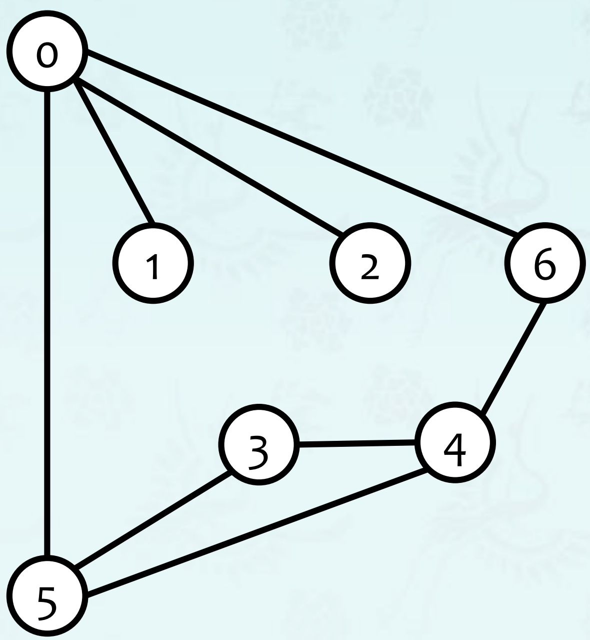
- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.

V-E lists



Adjacency lists



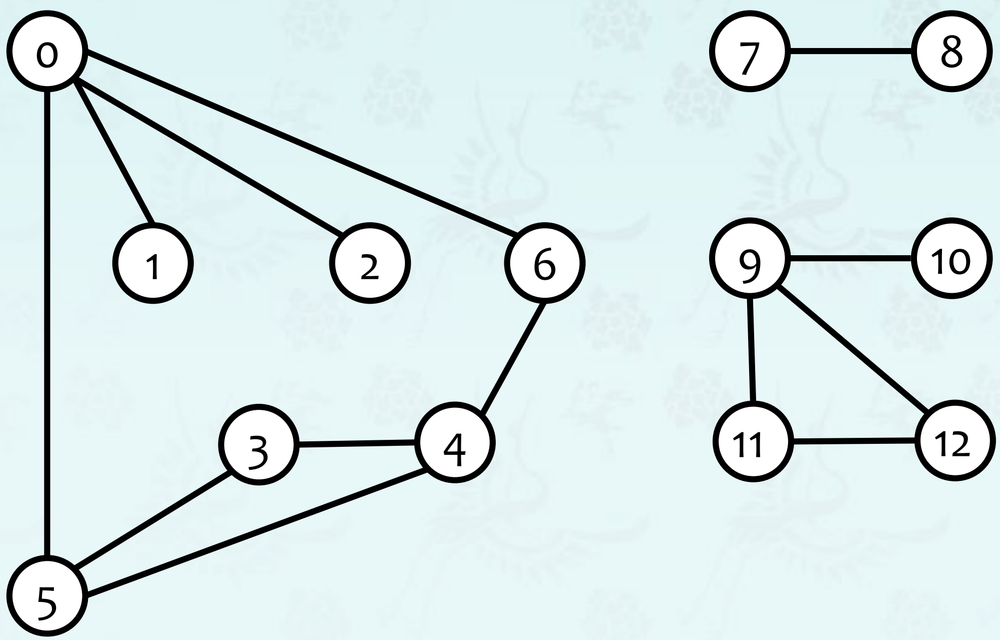


Graph g

Connected components

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



V-E lists

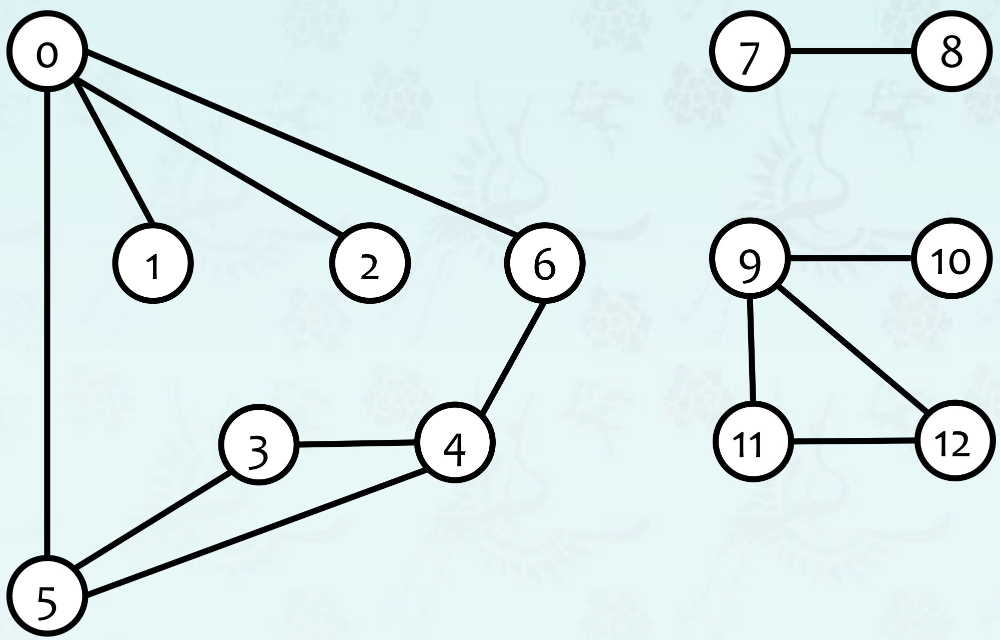
```
myG.txt
13    V
13    E
0  5
4  3
0  1
9  12
6  4
5  4
0  2
11  12
9  10
0  6
7  8
9  11
5  3
```

Graph g:

Connected components

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



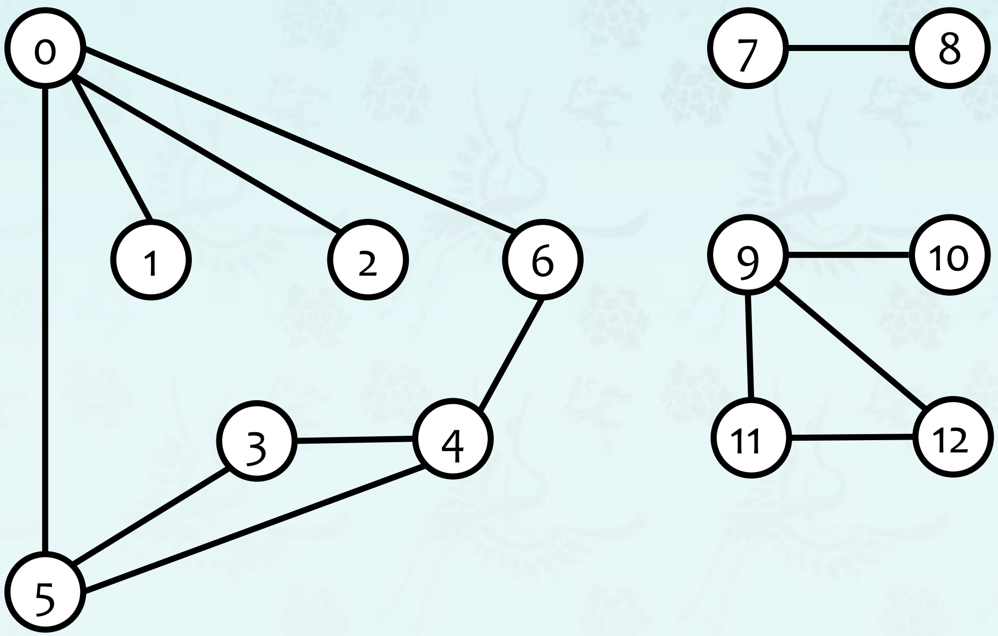
v	marked[]	id[]
0	F	–
1	F	–
2	F	–
3	F	–
4	F	–
5	F	–
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

Graph g:

Connected components

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



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

Done:

Finding Connected components – implementation in Java

```
public class CC {
  private boolean[] marked;
  private int[] id;
  private int count;

  public CC(Graph G) {
    marked = new Boolean[G.V()];
    id = new int[G.V()];
    for (int v= 0; v < G.V(); v++) {
      if (!marked[v]) {
        dfs(G, v);
        count++;
      }
    }

    public int count()
    public int id(int v)
    private void dfs(Graph G, int v)
  }
}
```

id[v]=id of component containing v

number of components

run DFS from one vertex in each component

see next slide

Finding Connected components – implementation in Java

```
public class count() {
  return count;
}
public int id(int v) {
  return id[v];
}
public void dfs(Graph G, int v) {
  marked[v] = true;
  id[v] = count;
  for (int w : G.adj[v]))
    if (!marked[w])
      dfs(G, w);
}
```

number of components

id of component containing v

all vertices discovered in the same call of dfs has the same id

ECE20010 Data Structures

Chapter 6

- *Introduction*
- *Graph API*
- *Elementary Graph Operations*
  - *DFS: Depth first search*
  - *BFS: Breadth first search*
  - ***CC: Connected Components***
- *HSet10 – graph.c*
  - ***implement DFS, BFS, CC and others***
  - ***submit it in dropbox***

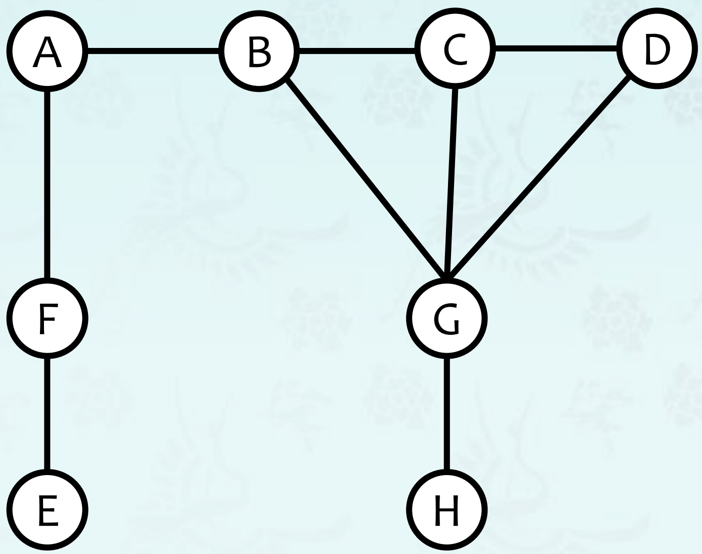
Major references:
1. Fundamentals of Data Structures by Horowitz, Sahni, Anderson-Freed,
2. Algorithms 4th edition - Part 1 & Part 2 by Robert Sedgewick and Kevin Wayne
3. Wikipedia and many resources available from internet

Prof. Youngsup Kim, idebtor@handong.edu, 2014 Data Structures, CSEE Dept., Handong Global University

DFS Exercise

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



adjacent list
A: B F
B: G C A
C: D G B
D: C G
E: F
F: E A
G: H B C D
H: G

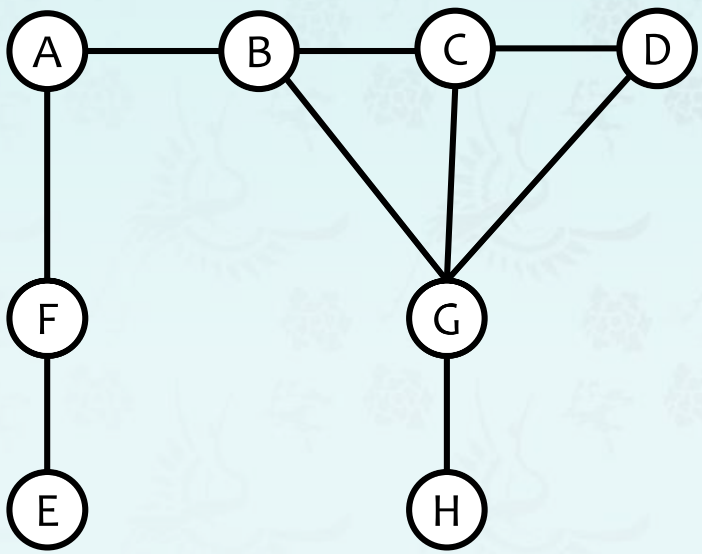
Graph g:

Hint: A B…?… F E

DFS Exercise

To visit a vertex v:

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v.



Graph g:

adjacent list

A: B F
B: G C A
C: D G B
D: C G
E: F
F: E A
G: H B C D
H: G

Hint: A B G H C D F E

```
dfs(A)
 dfs(B)
  dfs(G)
   dfs(H)
    check G
   H done
   check B
   dfs(C)
    dfs(D)
     check C
     check G
    D done
    check G
    check B
   C done
   check D
  G done
  check C
  check A
 B done
 dfs(F)
  dfs(E)
   check F
  E done
  check A
 F done
A done
check B
check C
check D
check E
check F
check G
check H
```