CMSC 132: Object-Oriented Programming II

UNDIRECTED GRAPHS

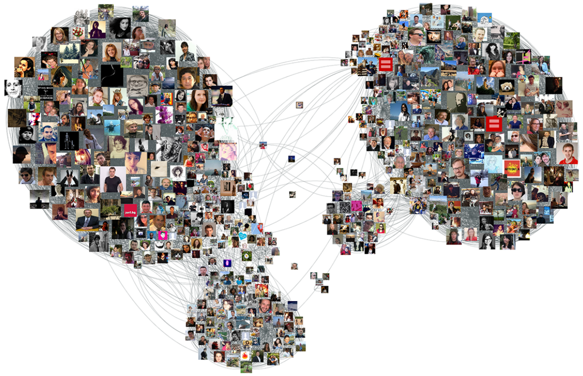
Graphs slides are modified from COS 126 slides of Dr. Robert Sedgewick.

Undirected Graphs

Graph: Set of vertices connected pairwise by edges.

Undirected Graphs

- Why study graph algorithms?
  - Thousands of practical applications.
  - Hundreds of graph algorithms known.
  - Interesting and broadly useful abstraction.
  - Challenging branch of computer science and discrete math.

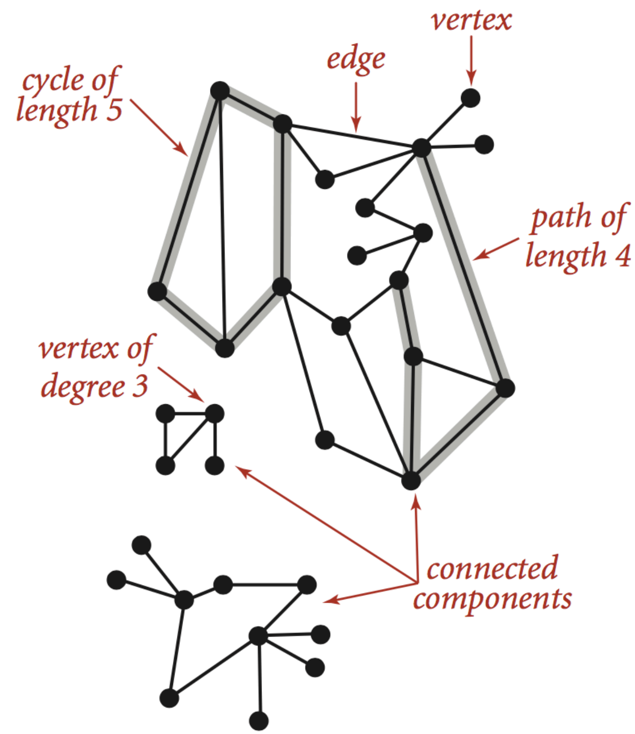


Graph Applications

Graph	Vertex	Edge
communication	Telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
chemical compound	molecule	bond

Graph Terminology

- Path:
  - Sequence of vertices connected by edges.
- Cycle
  - Path whose first and last vertices are the same.
- Two vertices are connected if there is a path between them.



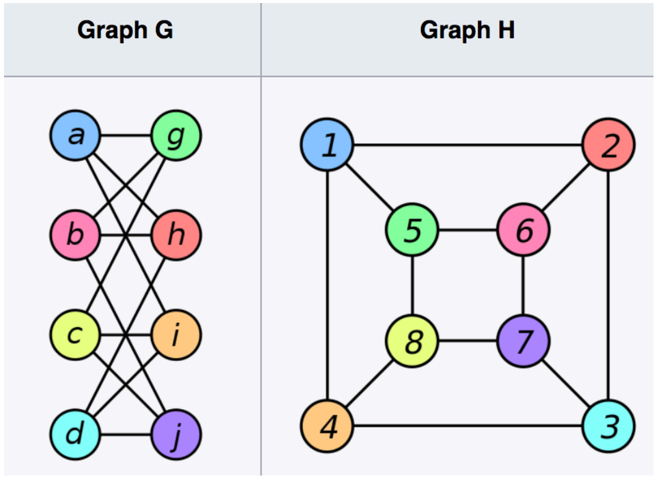
Some graph-processing problems

- Path:
  - Is there a path between s and t ?
- Shortest path.
  - What is the shortest path between s and t ?
- Cycle.
  - Is there a cycle in the graph?
- Euler tour.
  - Is there a cycle that uses each edge exactly once?
- Hamilton tour.
  - Is there a cycle that uses each vertex exactly once.
- Connectivity.
  - Is there a way to connect all of the vertices?

Some graph-processing problems

- MST.
  - What is the best way to connect all of the vertices?
- Biconnectivity.
  - Is there a vertex whose removal disconnects the graph?
- Planarity.
  - Can you draw the graph in the plane with no crossing edges
- Graph isomorphism.
  - Do two adjacency lists represent the same graph?

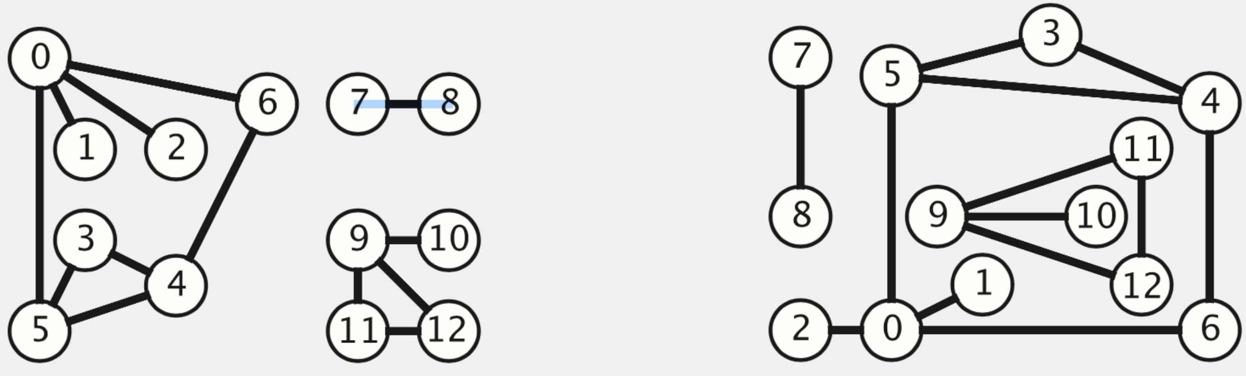
Challenge. Which of these problems are easy? difficult? intractable?



Graph representation

Graph drawing

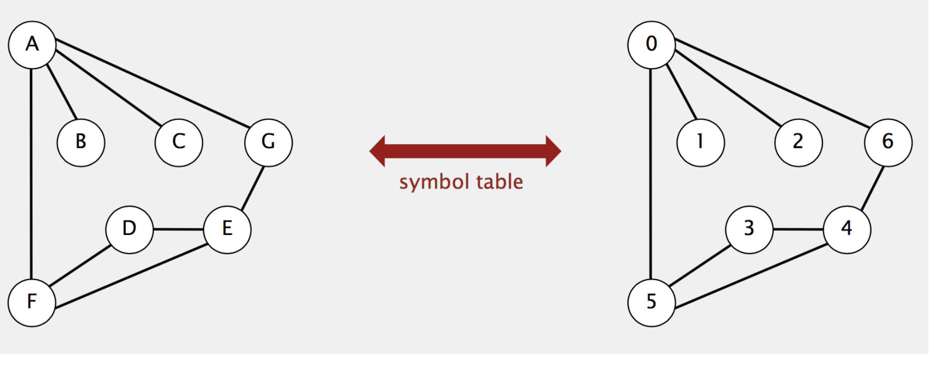
- Provides intuition about the structure of the graph.

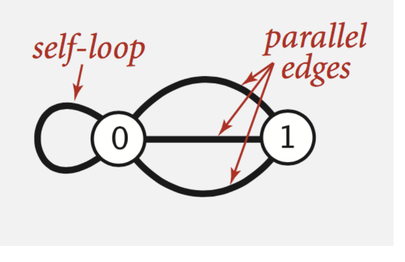


two drawings of the same graph

Graph representation

- Vertex representation:
  - use integers between 0 and $V - 1$.
- Applications: convert between names and integers with symbol table.





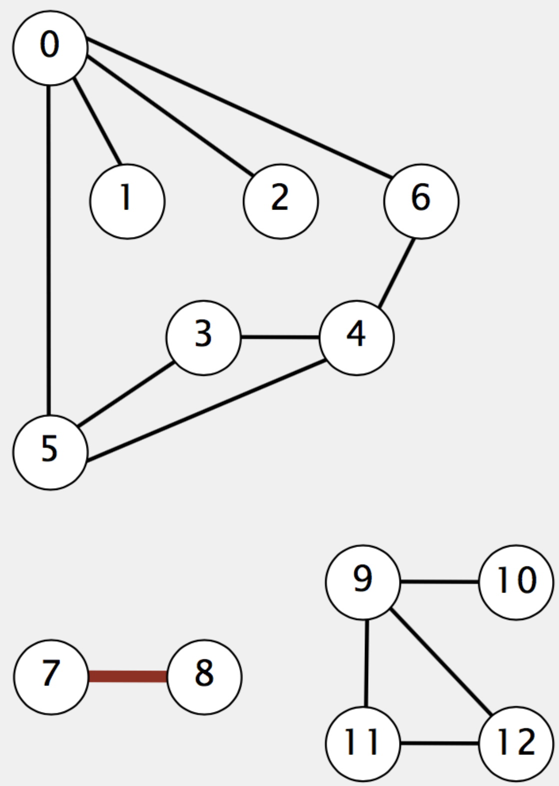
No self loop,
No parallel edges

Graph Class

```
public class Graph{
    Graph(int V)  //create an empty graph with V
    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
    String toString()  //string representation
}
```

Set-of-edges graph representation

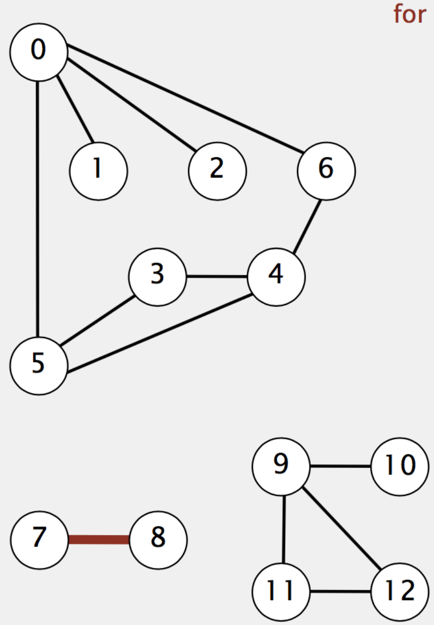
- Maintain a list of the edges (linked list or array).



0	1
0	2
0	5
0	6
3	4
3	5
4	5
4	6
7	8
9	10
9	11
9	12
11	12

Adjacency-matrix graph representation

- Maintain a two-dimensional *V*-by-*V* boolean array;
- for each edge *v*–*w* in graph:
  - adj[v][w] = adj[w][v] = true.

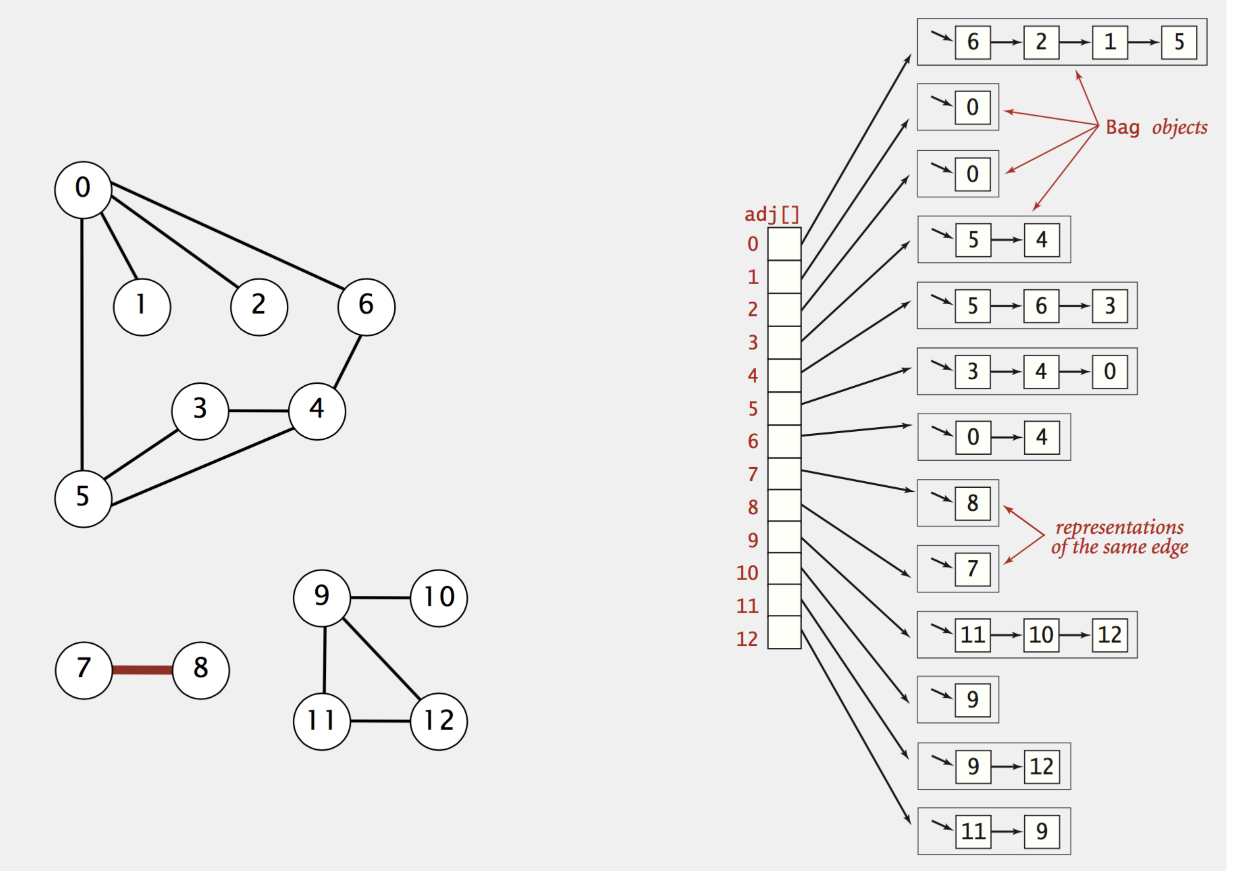


two entries for each edge

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

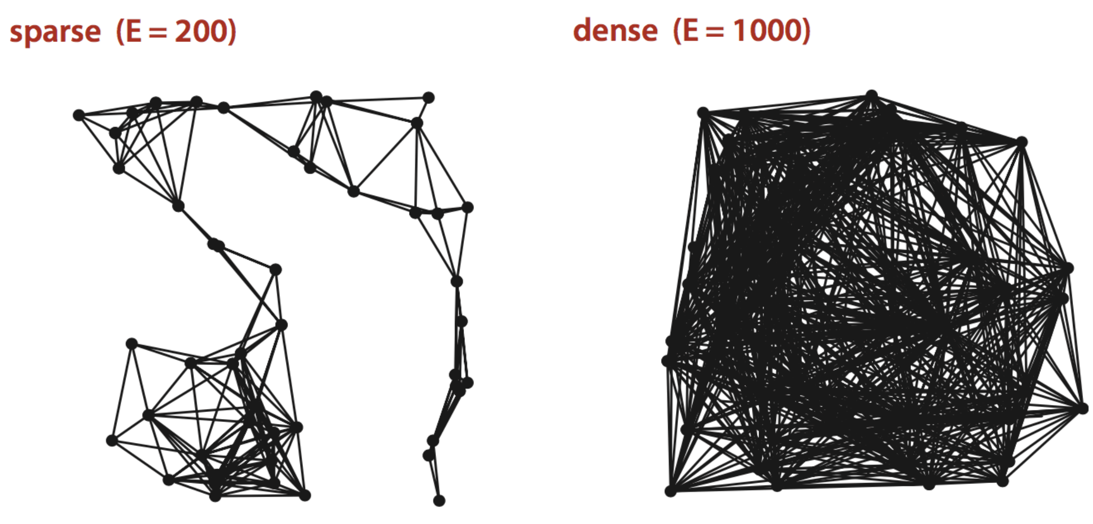
Adjacency-list graph representation

- Maintain vertex-indexed array of lists.



Graph representation

- In practice. Use adjacency-lists representation.
  - Algorithms based on iterating over vertices adjacent to *v*.
- Real-world graphs tend to be sparse.



Two graphs (V = 50)

huge number of vertices,
small average vertex degree

Graph representation

Comparisons of three different representations:

representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	E + V	1	degree(v)	degree(v)

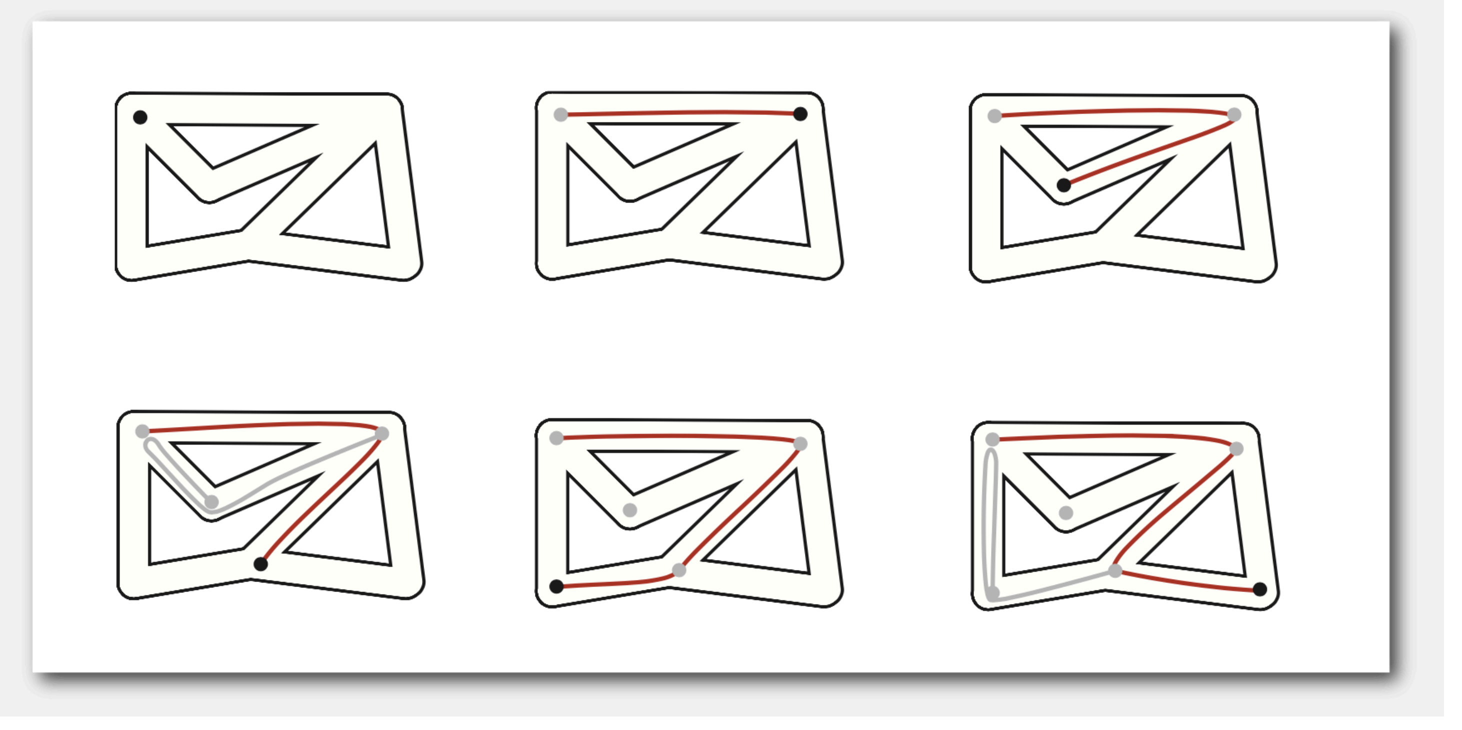
* disallows parallel edges

Adjacency-list graph representation: Java implementation

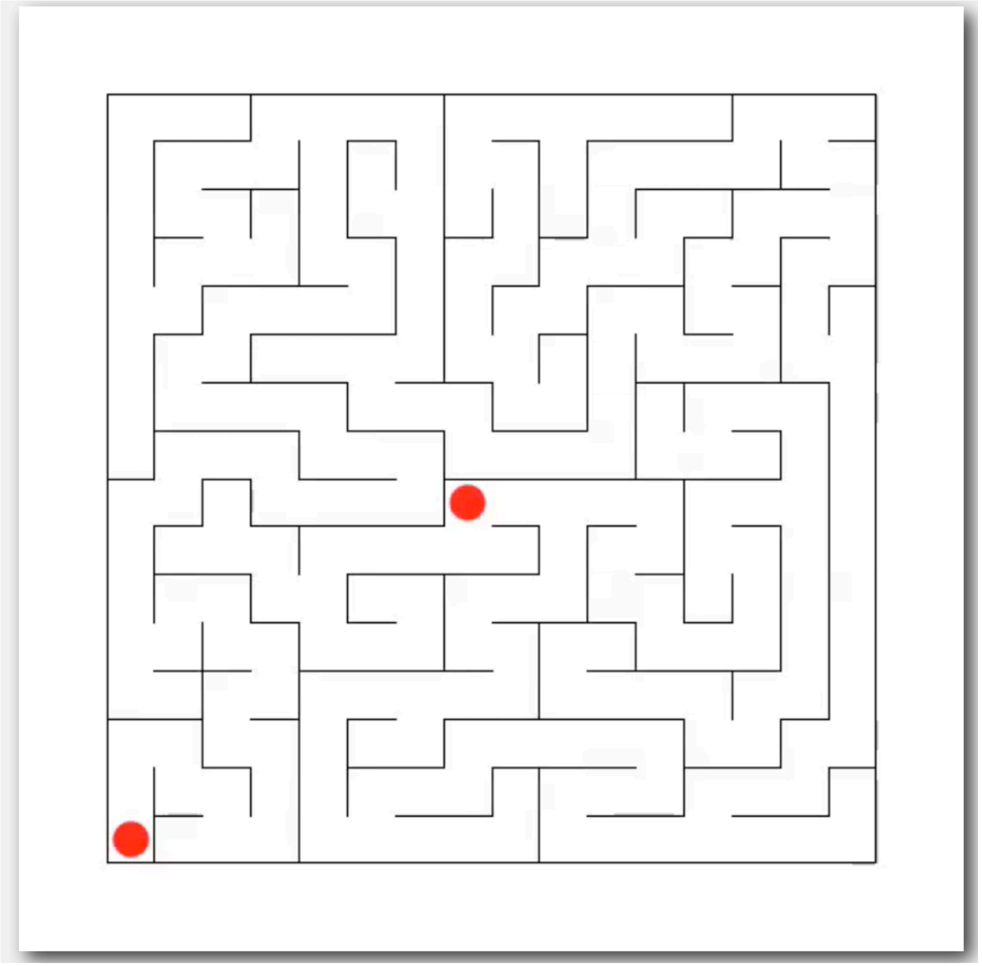
```
public class Graph{
  private final int V;
  private Bag<Integer>[] adj;
  public Graph(int V)  {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
      }
   public void addEdge(int v, int w) {
     adj[v].add(w);
     adj[w].add(v);
   }
    public Iterable<Integer> adj(int v) {
       return adj[v];
    }
}
```

Graph Algorithms: Depth First Search

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



Maze Exploration



Depth First Search

Goal. Systematically search through a graph. Idea. Mimic maze exploration.

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

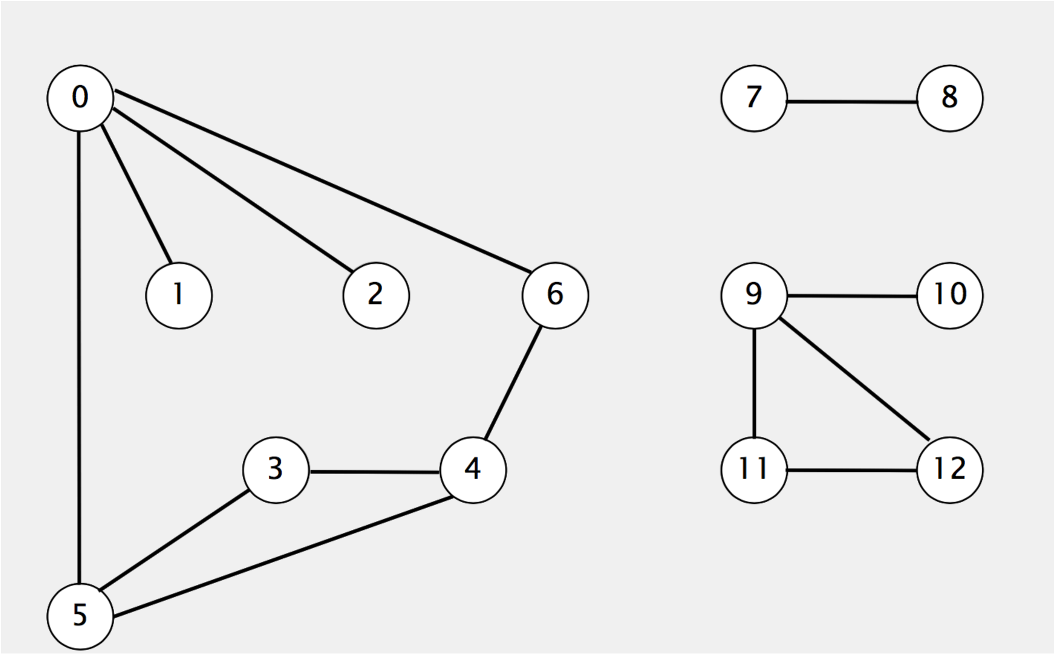
- Typical applications:
  - Find all vertices connected to a given source vertex.
  - Find a path between two vertices.

DFS Demo

To visit a vertex *v* :

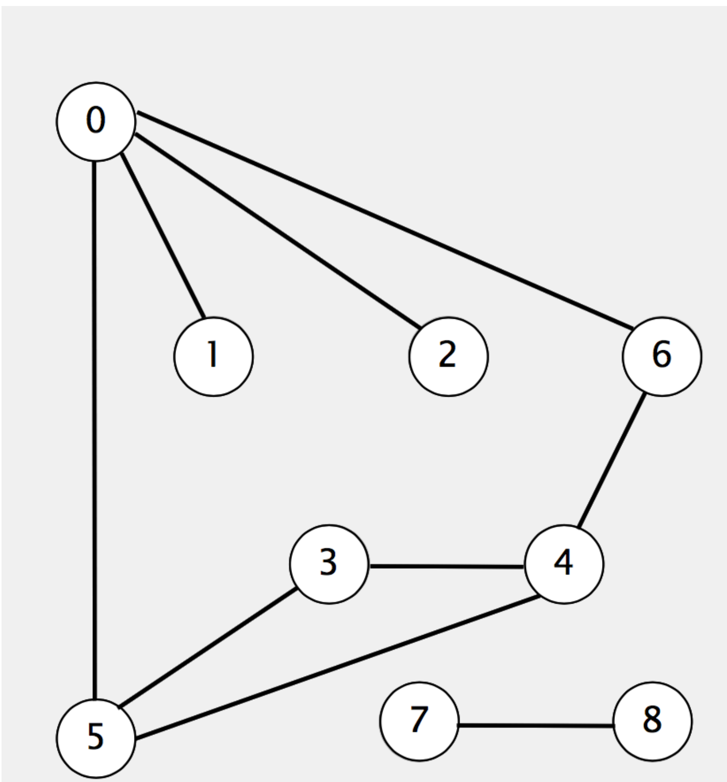
Mark vertex *v* as visited.

Recursively visit all unmarked vertices adjacent to *v*.



DFS Demo

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



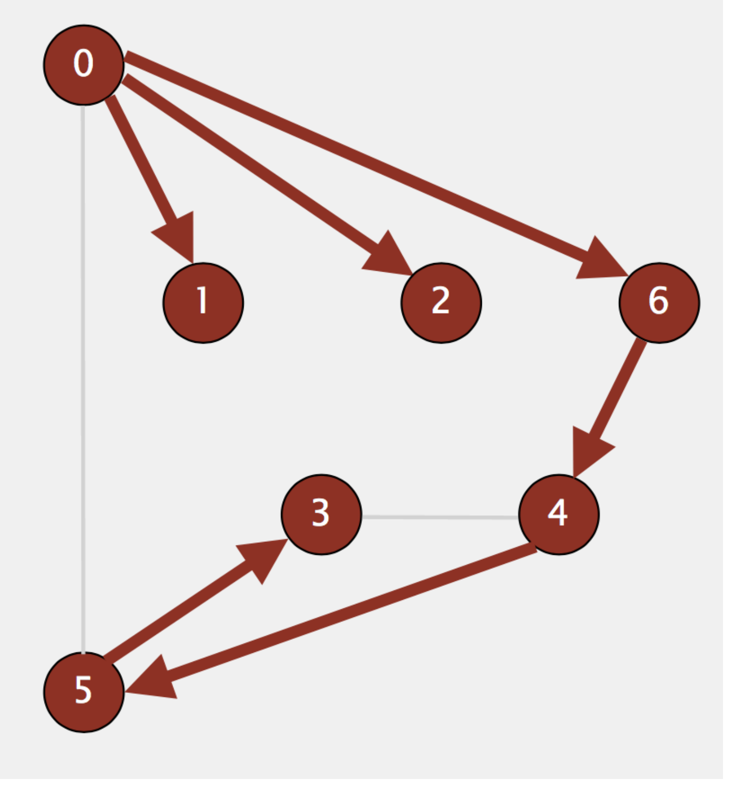
V	**marked[]**	**edgeTo[v]**
0		-
1		
2		
3		
4		
5		
6		
7		
8		

DFS Demo

To visit a vertex *v* :

Mark vertex *v* as visited.

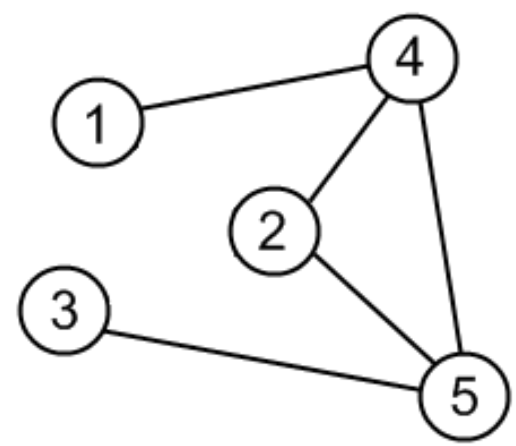
Recursively visit all unmarked vertices adjacent to *v*.



V	**marked[]**	**edgeTo[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	

Depth-first search

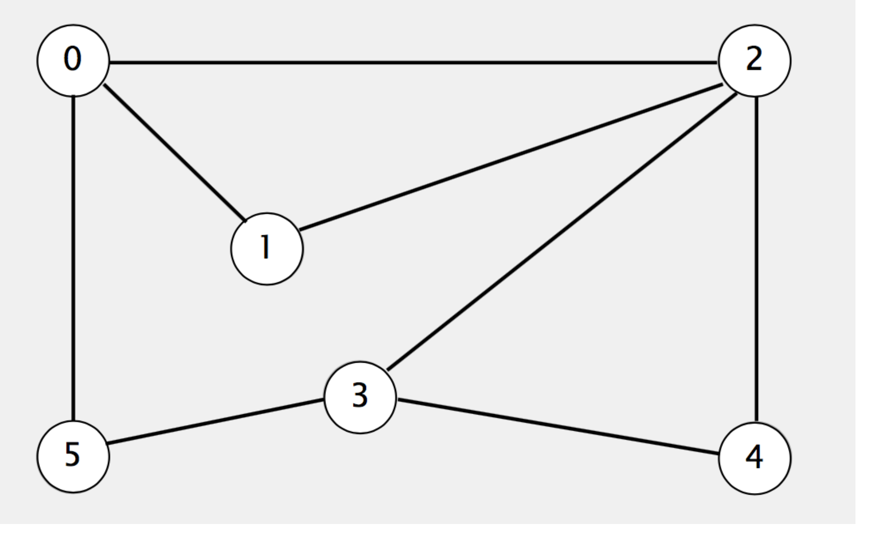
```
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] edgeTo;
  private int s;
  public DepthFirstSearch(Graph G, int s) {
     ...
  dfs(G, s);
  }
  private void dfs(Graph G, int v) {
     marked[v] = true;
     for (int w : G.adj(v))
        if(!marked[w]) {
            dfs(G, w);
            edgeTo[w] = v;
        }
  }
}
```



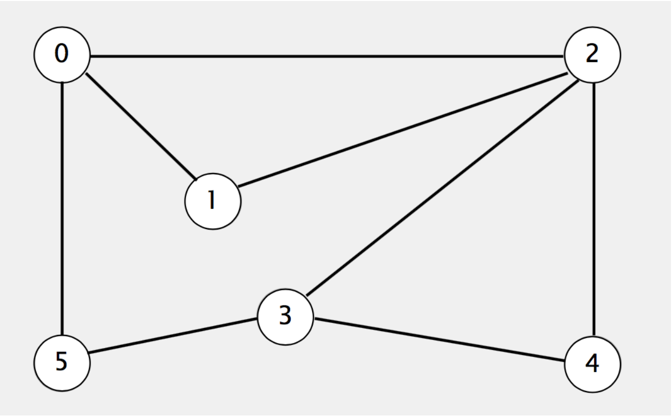
Breadth-first search (BFS)

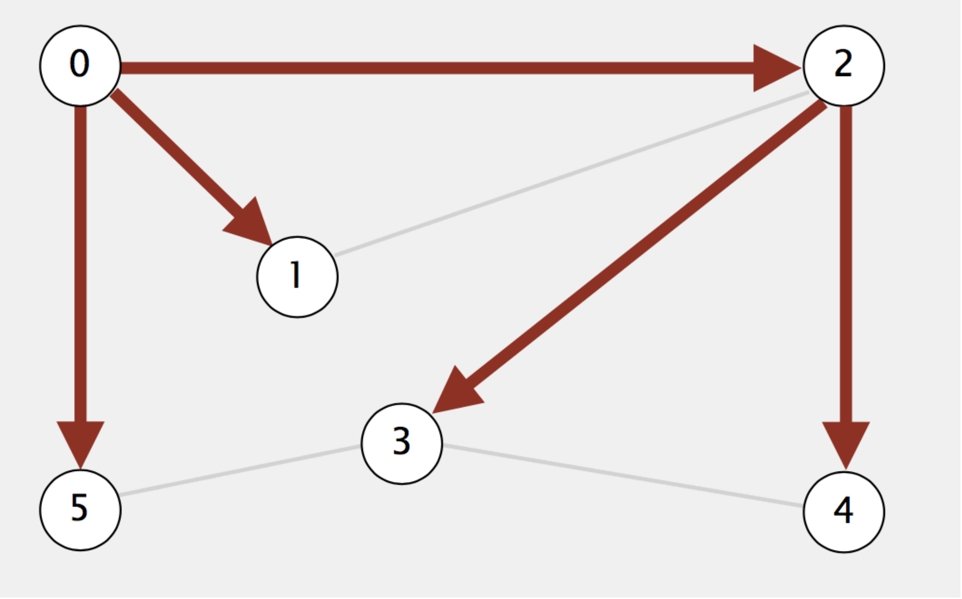
- BFS starts at a vertex and and explores the neighbor vertices first, before moving to the next level neighbors.

```
Repeat until queue is empty:
    Remove vertex v from queue.
    Add to queue all unmarked vertices
            adjacent to v and mark them.
```



Breadth-first search (BFS)

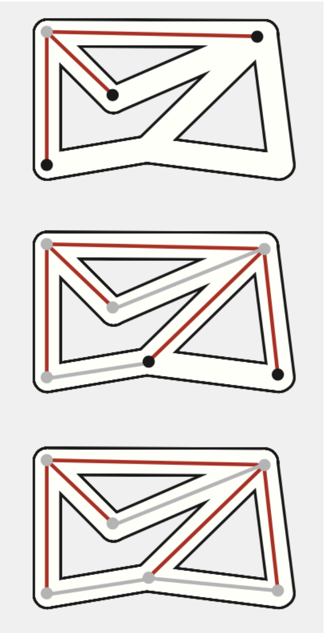




v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

Breadth-first search

Depth-first search: Put unvisited vertices on a stack.
Breadth-first search: Put unvisited vertices on a queue.
Shortest path: Find path from s to t that uses fewest number of edges.



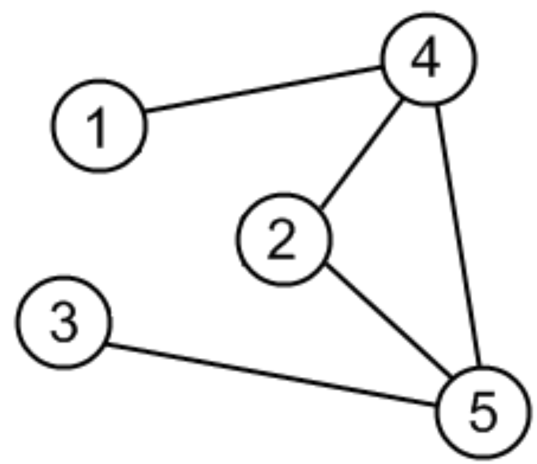
BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.
Repeat until the queue is empty:
- remove the least recently added vertex v
- add each of v's unvisited neighbors to the queue, and mark them as visited.

Intuition: BFS examines vertices in increasing distance from s.

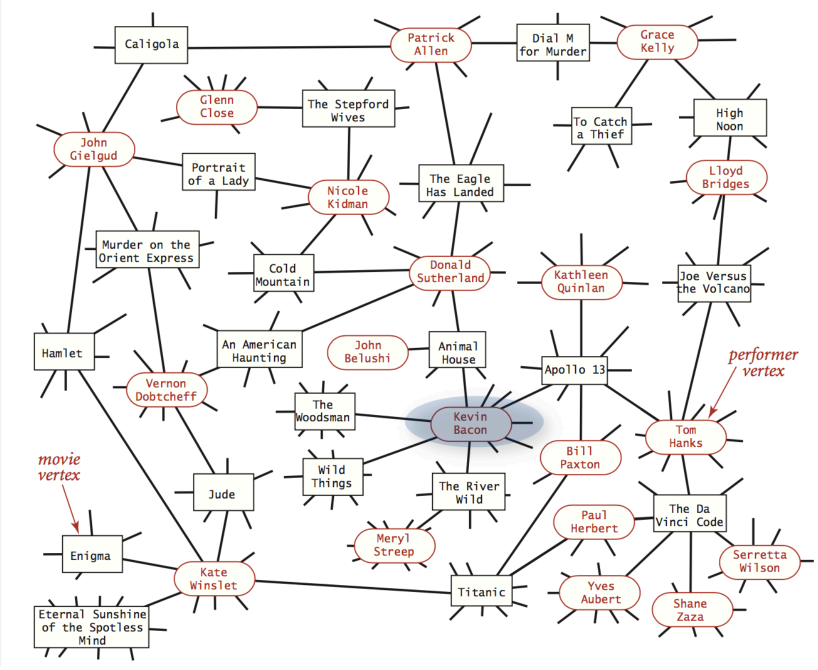
Breadth-first search

```
public class BreadthFirstPaths {
    private boolean[] marked;
    private int[] edgeTo;
    ...
    private void bfs(Graph G, int s) {
      Queue<Integer> q = new Queue<Integer>();
      q.enqueue(s);
      marked[s] = true;
      while (!q.isEmpty()) {
          int v = q.dequeue();
          for (int w : G.adj(v)) {
              if (!marked[w]){
                  q.enqueue(w);
                  marked[w] = true;
                  edgeTo[w] = v;
              }
          }
      }
    }
}
```



BFS Application: Kevin Bacon Number

- Kevin Bacon graph
  - Include one vertex for each performer and one for each movie.
  - Connect a movie to all performers that appear in that movie
  - Compute shortest path from *s* = Kevin Bacon.



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.