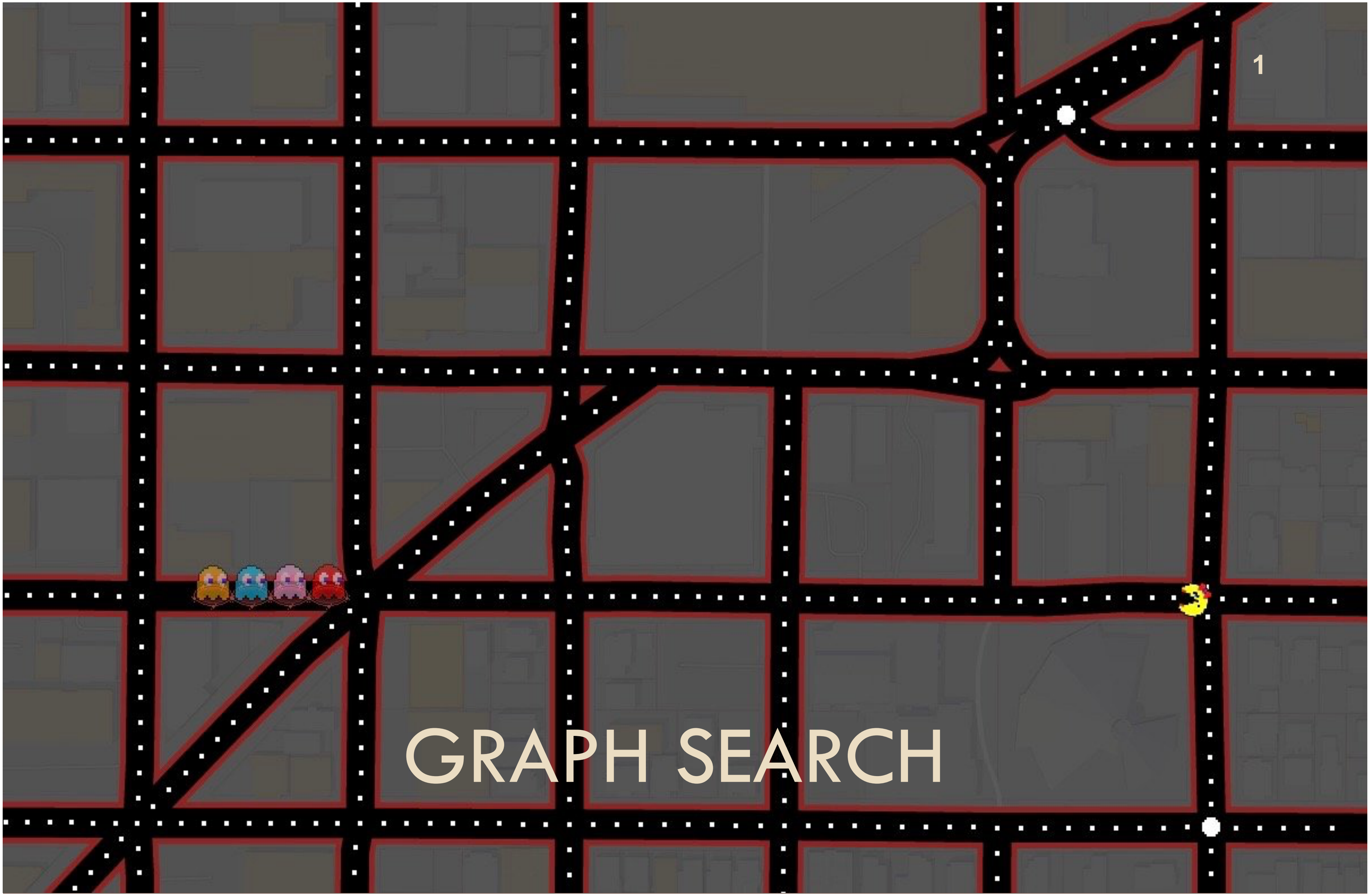


# GRAPH SEARCH

Lecture 18

CS 2110

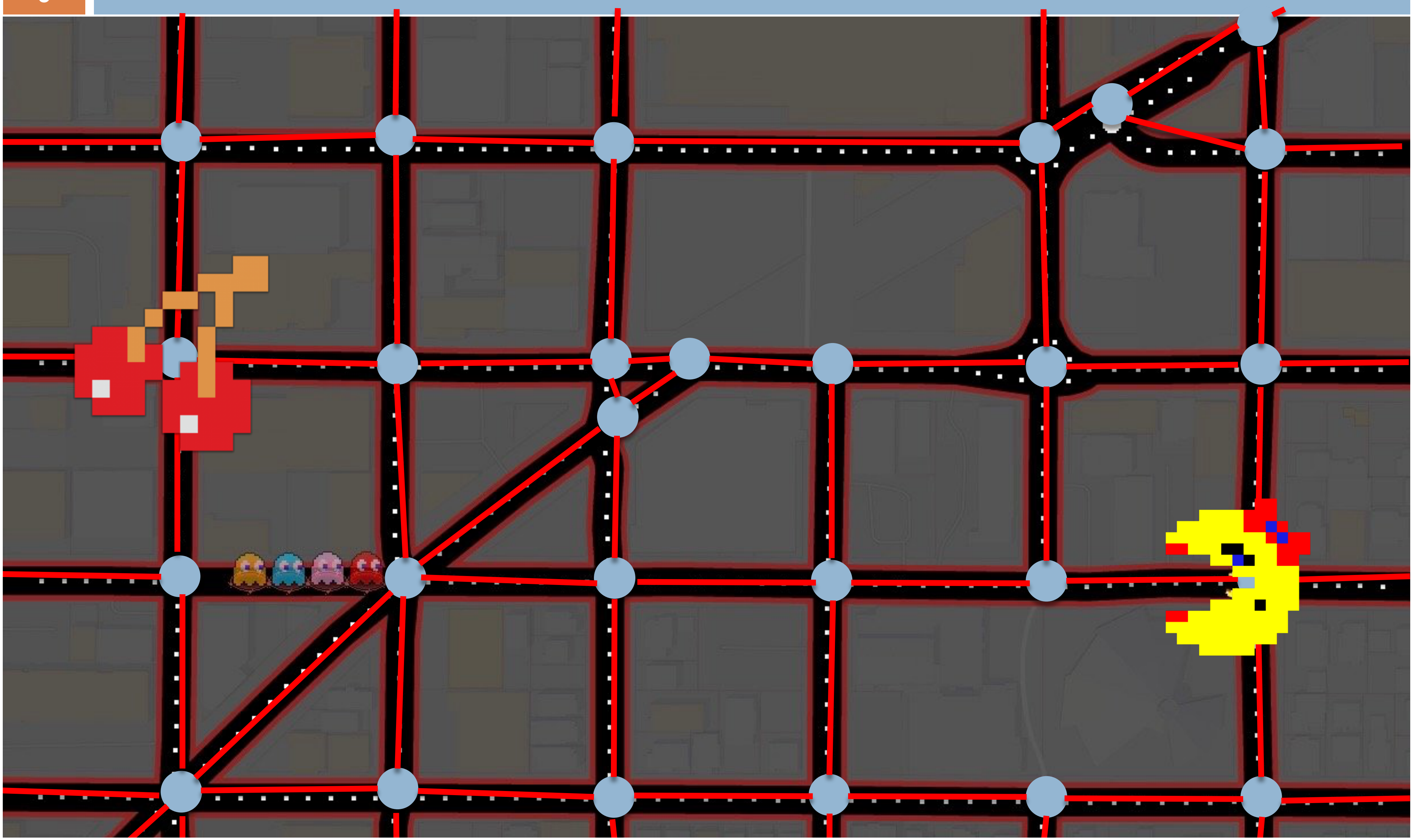
# Announcements



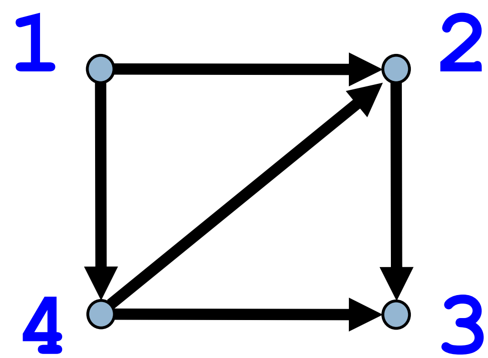
- TODO before next Tuesday:
  - Watch the tutorial on the shortest path algorithm
  - Complete the associated the Quiz

# Graphs

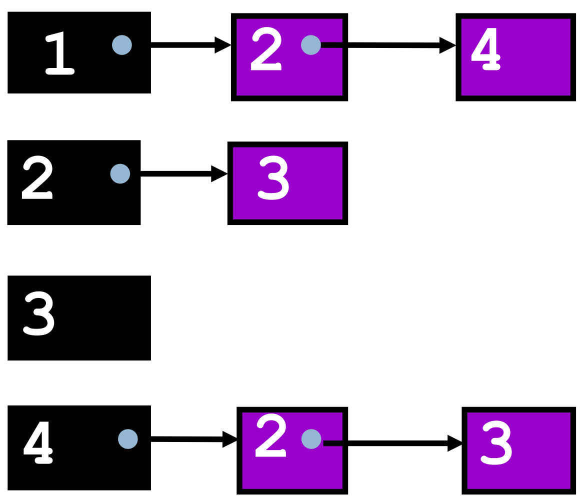




# Representing Graphs




Adjacency List




Adjacency Matrix

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 |
| 4 | 0 | 1 | 1 | 0 |

# Graph Interface

```
public interface Graph {

    /** Return the number of nodes in the graph */
    public int numNodes();

    /** Return a list of edges in the graph */
    public List<Pair> getEdges();

    /** Check whether an edge exists */
    public boolean hasEdge(int u, int v);

    /** Return a list of neighbors of n.
    * Precondition: 0 <= n < number of nodes */
    public List<Integer> getNeighbors(int n);

    /** Print the graph.
    * Precondition: the graph has < 100 nodes */
    public void printGraph();
}
```

# Pair Class

```java
/** An instance is an ordered pair of integers */
public class Pair {
       public int one;  // the ordered pair (one, two)
       public int two;

    /** Constructor: a pair of ints h and k. */
    public Pair(int h, int k) {
       one= h;
       two= k;
    }

    /** A representation (h, k) of this pair.*/
    public String toString() {
        return "(" + one + ", " + two + ")";
    }


}
```

# MatrixGraph Class

```
/** An instance is a graph maintained as an adjacency
matrix */
public class MatrixGraph implements Graph{
    public boolean[][] matrix; // adjacency matrix
    public int n;       // number of nodes
    public int m;       // number of edges


    /** A graph with n nodes numbers 0..n-1 and edges
     * given by edges.  */
    public MatrixGraph(int numNodes, Pair[] edges) {
        n= numNodes;
        m= edges.length;

        matrix= new boolean[n][n];
        for (Pair p : edges) {
            matrix[p.one][p.two]= true;
        }
    } ...
```

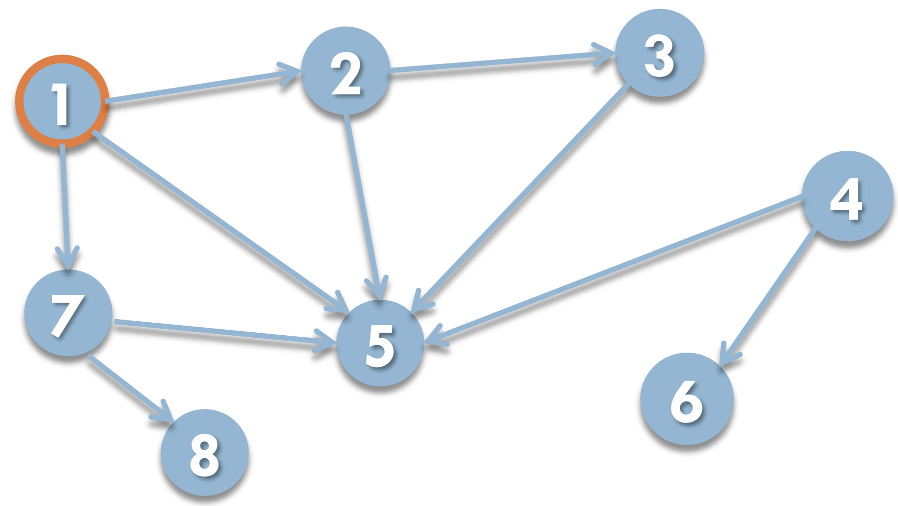
# Graph Algorithms

- Search
  - Depth-first search
  - Breadth-first search
- Shortest paths
  - Dijkstra's algorithm
- Spanning trees
  - Algorithms based on properties
  - Minimum spanning trees
  - Prim's algorithm
  - Kruskal's algorithm

# Search on Graphs

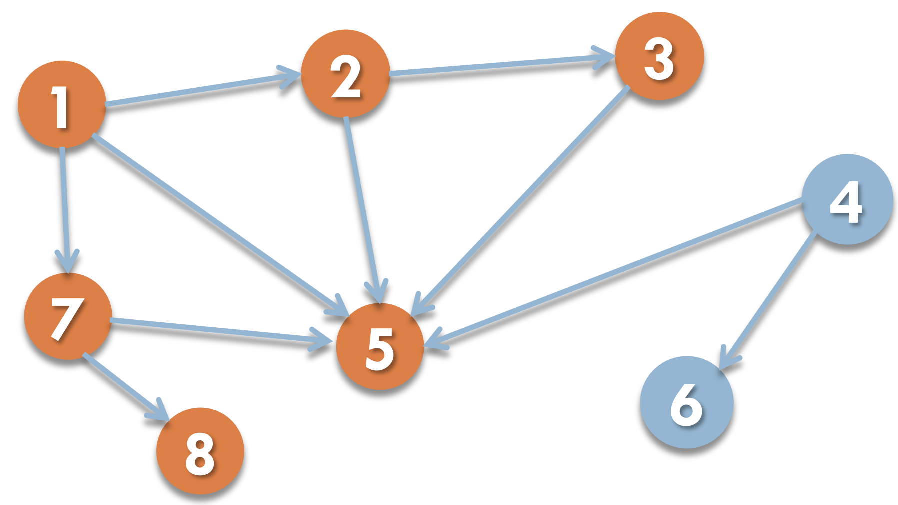


- Given a graph $(V, E)$ and a vertex $u \in V$
- We want to "visit" each node that is reachable from $u$




There are many paths to some nodes.

How do we visit all nodes efficiently, without doing extra work?

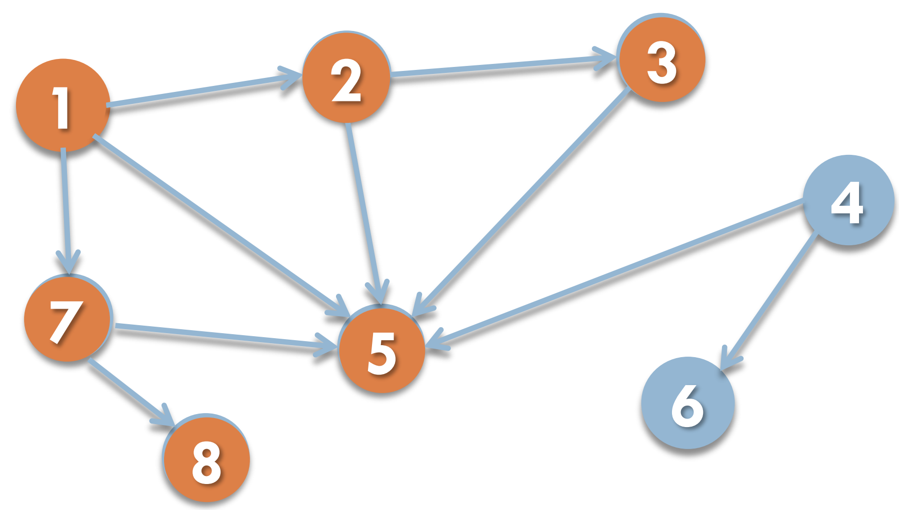


# Depth-First Search



Intuition: Recursively visit all vertices that are reachable along unvisited paths.

```
/** Visit all nodes reachable
on unvisited paths from u.
Precondition: u is unvisited.
*/
public static void dfs(int u) {
    mark u
    for all edges (u,v):
        if v is unmarked:
            dfs(v);
}
```




dfs(1) visits the nodes in this order: 1, 2, 3, 5, 7, 8

# Depth-First Search



Intuition: Recursively visit all vertices that are reachable along unvisited paths.

```
/** Visit all nodes reachable
on unvisited paths from u.
Precondition: u is unvisited.
*/
public static void dfs(int u) {
    mark u
    for all edges (u,v):
        if v is unmarked:
            dfs(v);
}
```

Suppose there are $n$ vertices that are reachable along unvisited paths and $m$ edges:

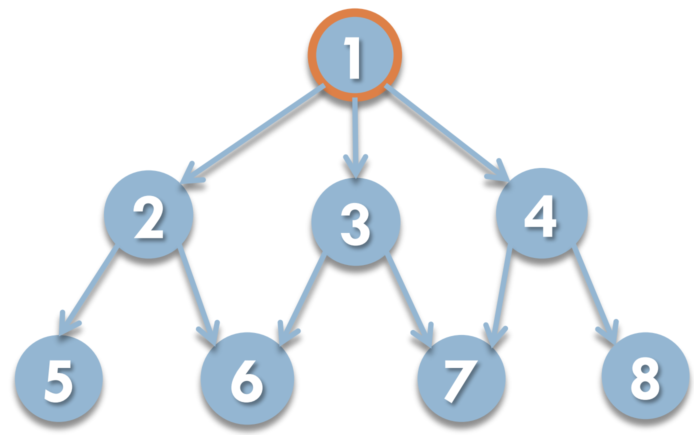
Worst-case running time? $O(n + m)$

Worst-case space? $O(n)$

# DFS Quiz



- In what order would a DFS visit the vertices of this graph? Break ties by visiting the lower-numbered vertex first.
  - 1, 2, 3, 4, 5, 6, 7, 8
  - 1, 2, 5, 6, 3, 6, 7, 4, 7, 8
  - 1, 2, 5, 3, 6, 4, 7, 8
  - 1, 2, 5, 6, 3, 7, 4, 8



# Depth-First Search in Java



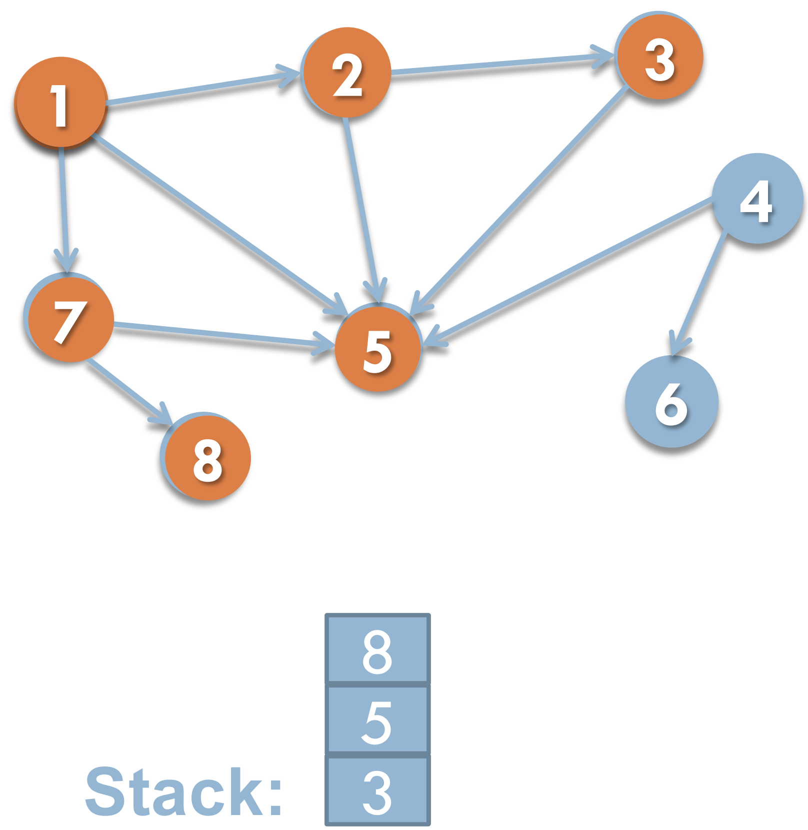
Eclipse!

# Depth-First Search Iteratively



Intuition: Visit all vertices that are reachable along unvisited paths from the current node.

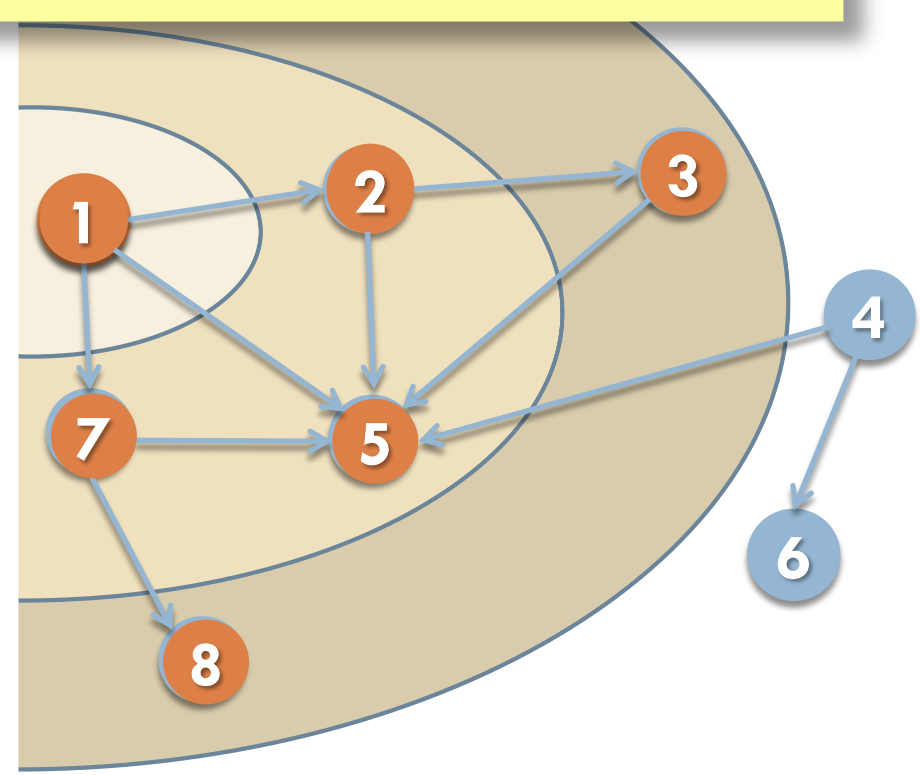
```
/** Visit all nodes reachable on
unvisited paths from u. */
public static void dfs(int u) {
    Stack s= new Stack
    s.push(u);
    while (s is not empty) {
        u= s.pop();
        if (u not visited) {
            visit u;
            for each edge (u, v):
                s.push(v);
        }
    }
}
```



# Breadth-First Search



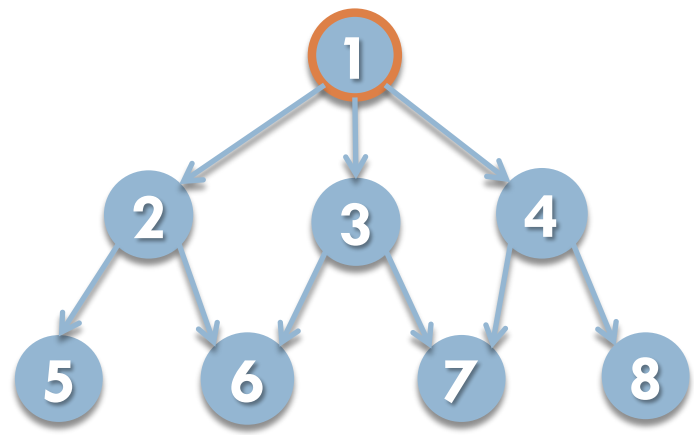
Intuition: Iteratively process the graph in "layers" moving further away from the source node.



# BFS Quiz



- In what order would a BFS visit the vertices of this graph? Break ties by visiting the lower-numbered vertex first.
  - 1, 2, 3, 4, 5, 6, 7, 8
  - 1, 2, 3, 4, 5, 6, 6, 7, 7, 8
  - 1, 2, 5, 3, 6, 4, 7, 8
  - 1, 2, 5, 6, 3, 7, 4, 8

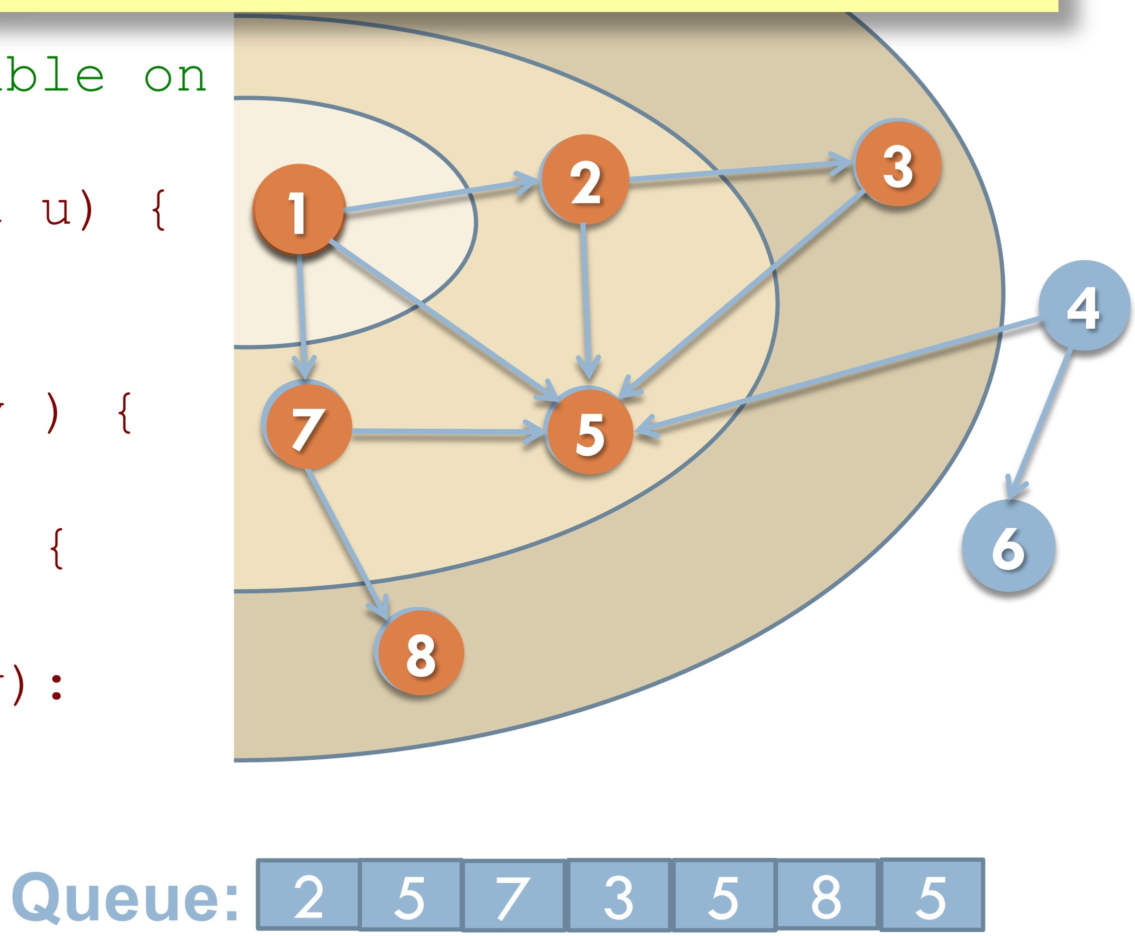


# Breadth-First Search



Intuition: Iteratively process the graph in "layers" moving further away from the source node.

```
/** Visit all nodes reachable on
unvisited paths from u. */
public static void bfs(int u) {
    Queue q= new Queue
    q.add(u);
    while ( q is not empty ) {
        u= q.remove();
        if (u not visited) {
            visit u;
            for each (u, v):
                q.add(v);
            }
        }
}
```



# Analyzing BFS



Intuition: Iteratively process the graph in "layers" moving further away from the source node.

```
/** Visit all nodes reachable on
unvisited paths from u. */
public static void bfs(int u) {
    Queue q= new Queue
    q.add(u);
    while ( q is not empty ) {
        u= q.remove();
        if (u not visited) {
            visit u;
            for each (u, v):
                q.add(v);
            }
        }
}
```

Suppose there are $n$ vertices that are reachable along unvisited paths and $m$ edges:

Worst-case running time? $O(n + m)$

Worst-case space? $O(m)$

# Comparing Search Algorithms



**DFS**

- Visits: 1, 2, 3, 5, 7, 8
- Time: $O(n+m)$
- Space: $O(n)$

**BFS**

- Visits: 1, 2, 5, 7, 3, 8
- Time: $O(n+m)$
- Space: $O(m)$