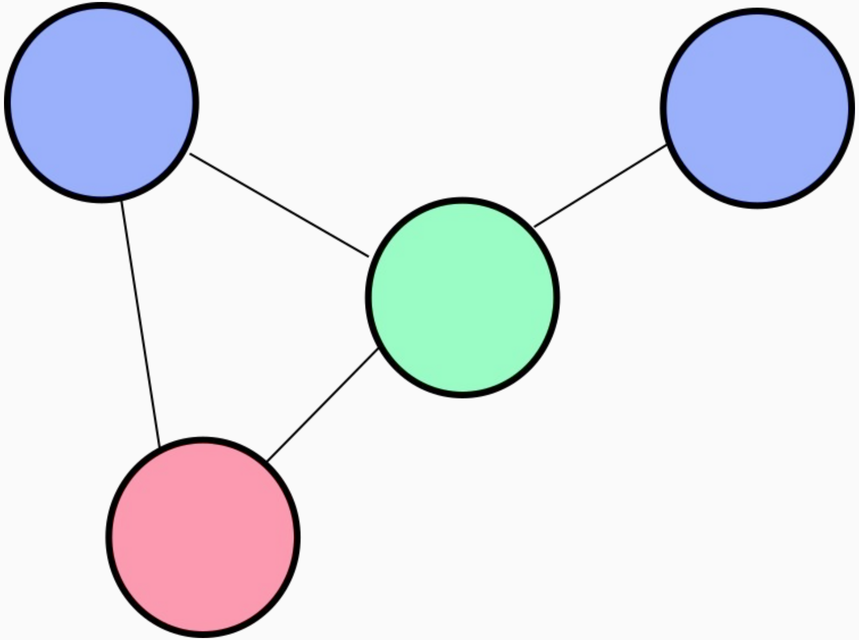
Bipartite Graphs and DFS

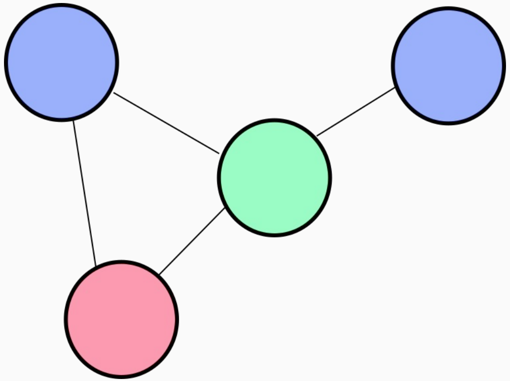
Coloring

- Assign a “color” to each node
- Sometimes we just assign numbers instead of actual color
  - eg: “color this node with 0, this node with 1”

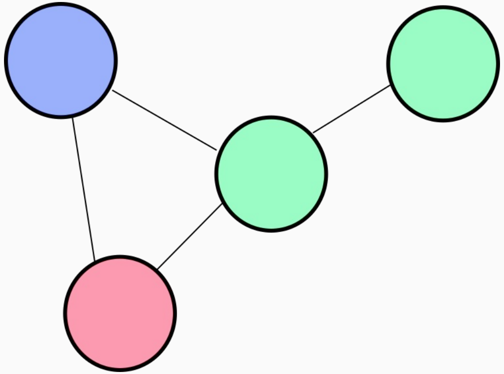


Coloring

- A valid node coloring means that **no neighbors are the same color**
- Valid:

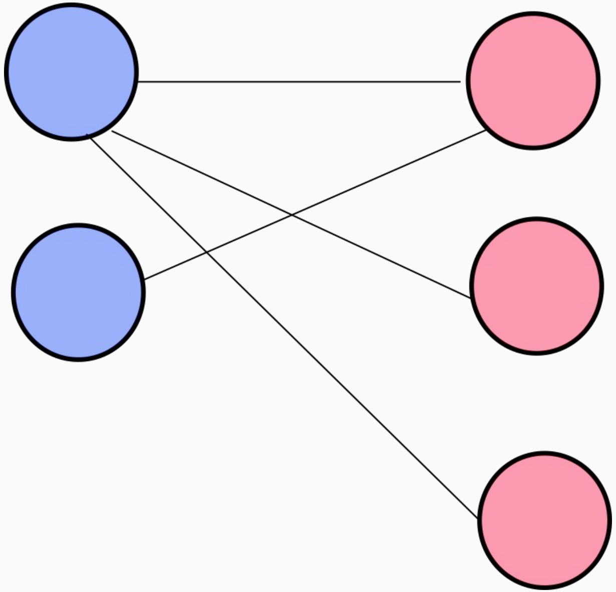


- Invalid:



Bipartite

- A bipartite graph is just a graph that has a valid 2-coloring
- This means we can divide the nodes into two sets:
    - Sometimes called left/right, X/Y, or A/B



Bipartite

- Also implies that there are no **odd-length** cycles: no triangles, etc.

Coloring in general

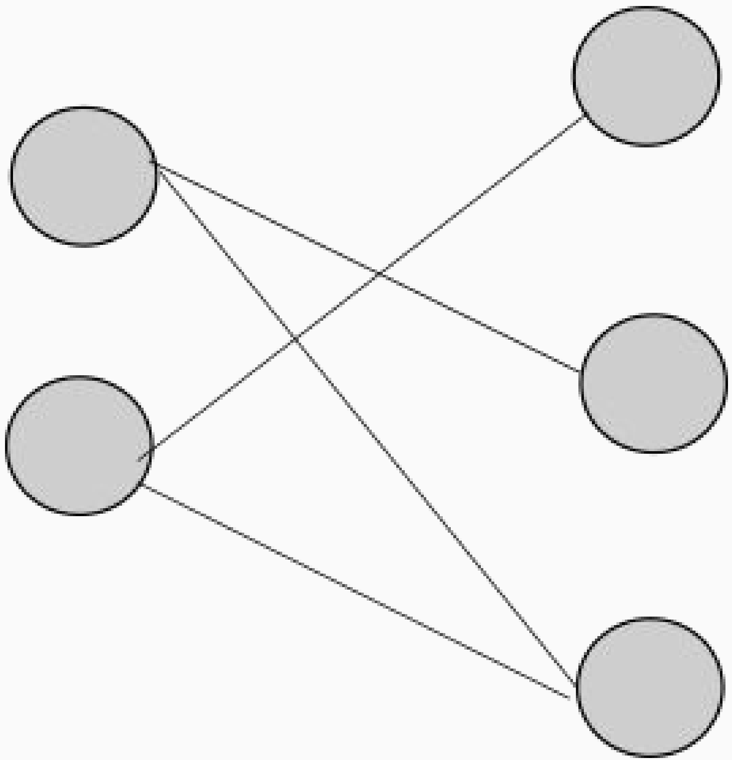
- Sometimes we want the minimum number of colors (the “chromatic number”)
- In general, finding the chromatic number is NP-Complete
    - ie: not feasible for a graph of any practical size

2-Coloring

- Can be done with a single graph traversal like BFS
- Any ideas?

2-Coloring

- Start with a node, color it “blue”
- Color all neighbors red
- Repeat as your traverse graph: attempt to color your neighbors the opposite of your color
- If you’ve already colored your neighbor, make sure the colors match
  - ie: if your neighbor is already “red” and so are you, this graph is not 2-colorable
- You either reach an invalid coloring point, or you’ve successfully 2-colored the graph.



Visualization (https://spruett.me/blog/static/color-yes.mp4)

Applications

- Many fundamental graph problems have special case algorithms on bipartite graphs
    - eg: max matching, vertex cover, etc.

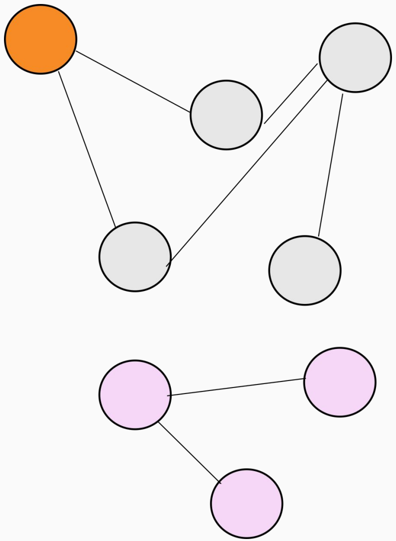
Applications

- Direct 2-coloring is useful for breaking things into two groups based on the edges

Implementation

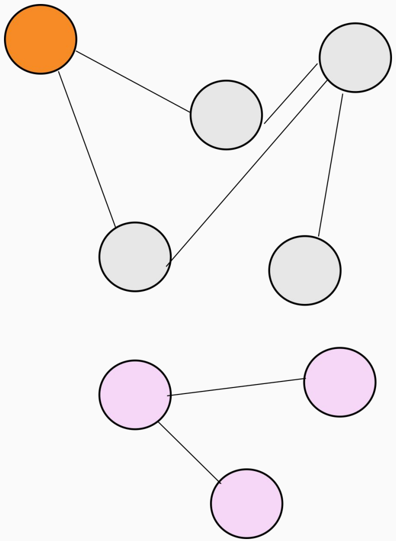
Graphs

- Last time we worked with a special case of graph: a connected graph
- In general, graphs may have many connected components
- We can call this graph “disconnected” because there is no path we can take from the orange node to any of the pink nodes



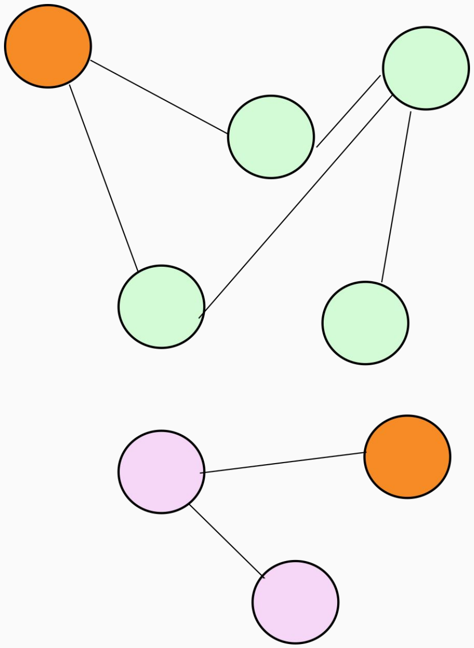
Graphs

- Last time we used BFS which started at a single node
- We want to test for bipartiteness by traversing the whole graph: where do we start?



Graphs

- Could BFS from orange until all are visited in that “connected component”, then BFS from another node
- Repeat until all nodes visited!

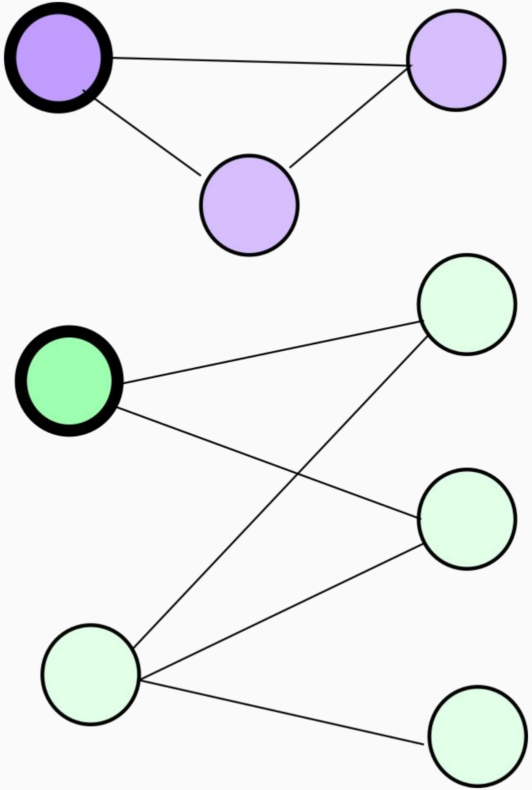


```java
static Set<Node> visited = ...;
static void traverseAll() {
    for (Node node : graph) {
        if (!visited.contains(node)) {
            traverseFrom(node);
            // could use BFS, DFS, etc.
        }
    }
}
```

Implementation

- How does this apply to 2-coloring?

Sample disconnected graph



- Is the green component bipartite?

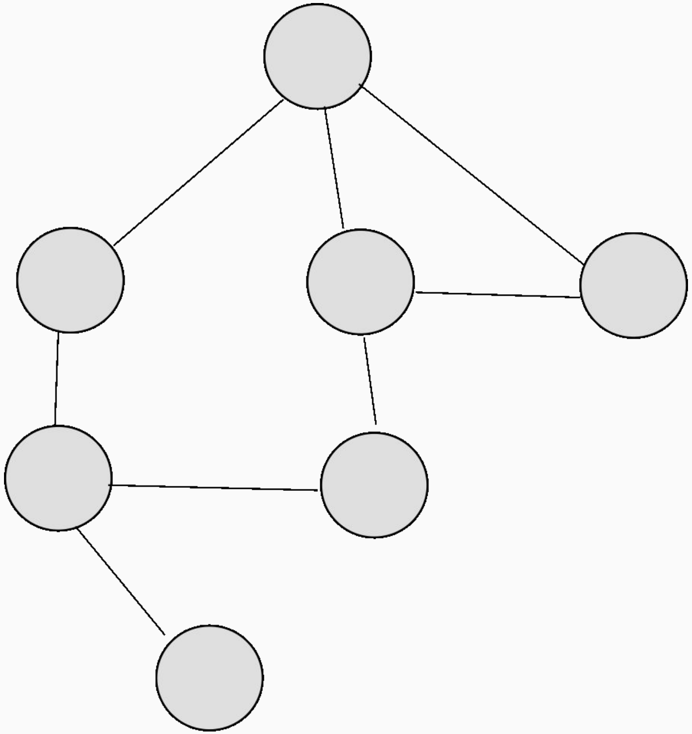
- Is the purple component bipartite?

Depth-first Search (DFS)

DFS

- BFS processed nodes in increasing order of distance from the start
- This expands out “breadth first” from the starting node
- DFS proceeds straight down “depth first”

BFS



DFS

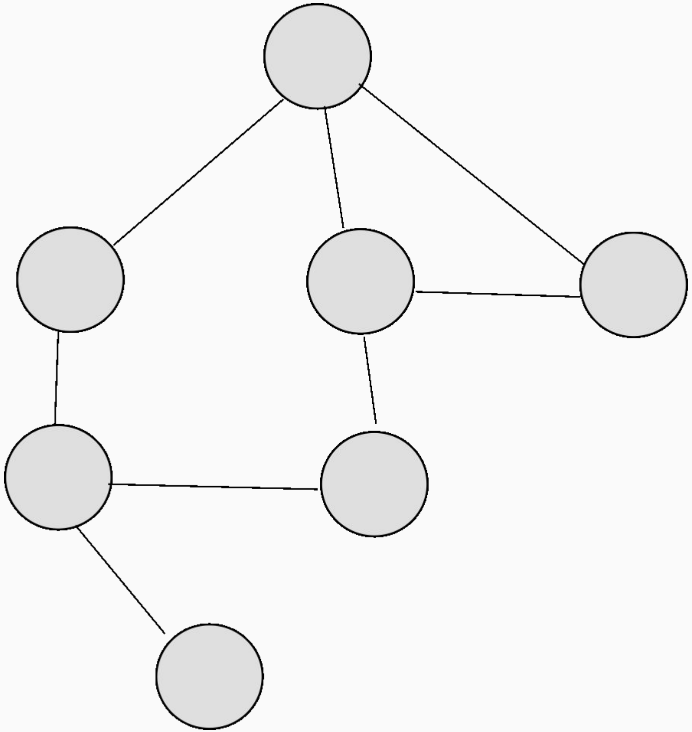


Illustration of BFS and DFS

Implementation differences

- DFS only changes the order in which we process nodes
- Any guesses as to what data structure?

Implementation differences

- We can implement an iterative DFS exactly the same as BFS but change the queue to a stack
- In Java, we frequently avoid java.util.Stack, and instead use an ArrayDeque (which allows us to use it like a stack or a queue)
- We can use deque.offerLast/deque.pollLast to mimic a stack

```java
Deque<Integer> queue = new ArrayDeque<Integer>();
queue.offerLast(start);
visited.add(start);

while (!queue.isEmpty()) {
    int current = queue.pollLast();
    for (int adj : graph.get(current)) {
        if (visited.contains(adj)) {
            // ...
        }
        else {
            queue.offer(adj);
            visited.add(adj);
        }
    }
}
```

Iterative DFS code (https://spruett.me/blog/static/code/DFSTemplate.java.html)

Recursion

- Show of hands for passed 2114 already?

Recursive DFS

- If you’ve ever done a recursive tree traversal of any kind, you’ve done recursive DFS

```java
static ... dfs(Node v, ...) {
    if (visited.contains(v)) {
        return; // possibly return something
    }
    visited.add(v);
    // calculate with v based on parameters
    for (Node a : graph.get(v)) {
        // use return value of children to do something
        x = dfs(a, ...);
    }
    // return computation based on v or children
}
```

Sample recursive DFS outline (https://spruett.me/blog/static/code/RecurDFS.java.html)

Recursive DFS

- The idea is that the call stack replaces explicit stack (or ArrayDeque) of the iterative version
- Useful because you can now use parameters and return values!
- Generally more concise than iterative BFS or iterative DFS
- Be careful with extremely large graphs and the runtime stack

```java
static int sumDFS(Node v) {
    if (visited.contains(v)) {
        return 0; // return 0 because already counted this node
    }
    visited.add(v);
    int sum = v.value;
    for (Node a : graph.get(v)) {
        // sum over children (possibly ignoring visited ones)
        sum += sumDFS(a);
    }
    return sum;
}
```

DFS or BFS?

- How do you decide?
- Sometimes you’ll need to use BFS
  - ie: for shortest paths
- Sometimes you’ll be implementing an algorithm based on DFS
  - ie: bridge finding, BCC, SCC
- Sometimes you’ll only need to process the graph: order doesn’t matter
  - Pick whichever you think of first, or you think you can implement better/without bugs/faster