Graphs
Lecture Question

Task: Determine if two nodes connected

• In the week9.Graph class
  • Write a method named areConnected that takes two node indices (Ints) and determines if the two nodes are connected in the graph
  • Return true if they are connected, false if they are not
Data Structures: Review

- **Sequential Data Structures**
  - Elements stored in a specific order
  - Ex: Array, List

- **Key-Value Store**
  - Stores pairs of elements with no particular order
  - Each key is associated with one value
  - Ex. Map, Dictionary, Object

- **Tree**
  - Non-linear structure
  - Each element can be associated with multiple other elements

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Node</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>20</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;cse&quot;</td>
<td>20</td>
</tr>
<tr>
<td>&quot;mga&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;geo&quot;</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>3</td>
</tr>
</tbody>
</table>
• How do we store data with multiple interconnected associations?

• A [station, intersection, city] can have multiple connections
Let's use trees
Start with UCLA as the root
Recursively add all children

Oops
We have duplicates in our data structure
Data Structures

- Let's try again
- When we try to add a duplicate, add a reference to the existing node
Graphs

- This is a graph
- Similar to a tree, except cycles are allowed
  - Cycle: Can "travel" from a node back to itself without backtracking
Graphs

- Because of the cycles, our tree traversals will get stuck in infinite recursion
- No leaves to terminate the recursion
Graphs

- We'll need a new way of representing this data structure and new algorithms to work with the data.
- Store the nodes and edges.
Graphs - Nodes and Edges

- **Node**: Each data element is stored in a node, similar to linked lists and trees

- **Edge**: A connection between two nodes
Graphs - Adjacency List

- A map of nodes to all nodes connected to it through an edge
- This is how we'll represent graphs
When creating a graph, we'll assign each node a unique ID as an int

- Allows nodes with identical values, but different IDs

```scala
class Graph[A] {
  var nodes: Map[Int, A] = Map()
  var adjacencyList: Map[Int, List[Int]] = Map()

  def addNode(index: Int, a: A): Unit = {
    nodes += index -> a
    adjacencyList += index -> List()
  }

  def addEdge(index1: Int, index2: Int): Unit = {
    adjacencyList += index1 -> (index2 :: adjacencyList(index1))
    adjacencyList += index2 -> (index1 :: adjacencyList(index2))
  }
}
```
Graphs - Adjacency List

- IDs for each node are arbitrary as long as they are unique
- Methods will work with IDs
- Values are only accessed when needed

```scala
object GraphExample {
  def main(args: Array[String]): Unit = {
    val graph: Graph[String] = new Graph()
    graph.addNode(0, "UCLA")
    graph.addNode(1, "STANFORD")
    graph.addNode(2, "SRI")
    graph.addNode(3, "UCSB")
    graph.addNode(4, "RAND")
    graph.addNode(5, "UTAH")
    graph.addNode(6, "SDC")
    graph.addNode(7, "MIT")
    graph.addNode(8, "BBN")
    graph.addNode(9, "LINCOLN")
    graph.addNode(10, "CARNegie")
    graph.addNode(11, "HARVARD")
    graph.addNode(12, "CASE")
    graph.addEdge(0, 1)
    graph.addEdge(0, 2)
    graph.addEdge(0, 3)
    graph.addEdge(0, 4)
    graph.addEdge(1, 2)
    graph.addEdge(2, 3)
    graph.addEdge(3, 4)
    graph.addEdge(2, 5)
    graph.addEdge(5, 6)
    graph.addEdge(5, 7)
    graph.addEdge(4, 8)
    graph.addEdge(7, 8)
    graph.addEdge(7, 9)
    graph.addEdge(9, 12)
    graph.addEdge(12, 10)
    graph.addEdge(10, 11)
    graph.addEdge(11, 8)
  }
}
```
A path is a sequence of nodes where each pair of adjacent nodes are connected by an edge.

["UCLA", "SRI", "UTAH", "MIT", "BBN", "RAND"] is a path in this graph.

["SRI", "UTAH", "BBN"] is not a path since UTAH and BBN are not connected by an edge.
Breadth-First Search (BFS)
Connected Component

- This graph is connected
- There exists a path between any 2 nodes in the graph
Connected Component

• What if a few connections are broken?
• How can we tell if two nodes are connected?
Connected Component

• We could verify manually for this graph
• But the Internet has gotten a little bigger over time
• Need to code an algorithm to solve this for us
The Algorithm: Breadth-First Search (BFS)

- Choose a starting node
- Continuously explore connected nodes
BFS

- Choose a starting node
BFS

• Explore all nodes connected to the striating node
BFS

- Repeatedly explore nodes that were visited in the last round
BFS

- Repeat until no new nodes are added
- Never visit a node twice
BFS

- Use a queue to track the order of nodes to visit
- Start with starting node in the queue
- When visiting a node, add all unexplored neighbors to the queue
- Visit neighbors of the node at the front of the queue until the queue is empty

```scala
def bfs[A](graph: Graph[A], startID: Int): Unit = {
  var explored: Set[Int] = Set(startID)
  val toExplore: Queue[Int] = new Queue()
toExplore.enqueue(startID)
  while (!toExplore.empty()) {
    val nodeToExplore = toExplore.dequeue()
    for (node <- graph.adjacencyList(nodeToExplore)) {
      if (!explored.contains(node)){
        println("exploring: " + graph.nodes(node))
toExplore.enqueue(node)
        explored = explored + node
      }
    }
  }
}
```
Connectivity

- If you start at nodeA and explore nodeB during the algorithm
- nodeA and nodeB are connected

- For the lecture question and last HW you'll need to modify/expand the provided BFS code
Lecture Question

Task: Determine if two nodes connected

• In the week9.Graph class

• Write a method named areConnected that takes two node indices (Ints) and determines if the two nodes are connected in the graph

• Return true if they are connected, false if they are not