Today’s Outline

- Graphs
  - Reachability
  - Graph Coloring
    - Lab10
[TAP] Sum of degrees

- Let $\text{deg}(v)$ be the degree of a vertex $v$. Is the following statement true?
- For any graph $G = (V,E)$

\[
\sum_{v \in V} \text{deg}(v) = 2 \mid E \mid
\]

where $|E|$ is the number of edges in $G$

\[
\begin{align*}
\text{in } \text{deg}(v) &= 1 \\
\text{out } \text{deg}(v) &= 1 \\
\text{deg}(v) &= 2
\end{align*}
\]
Distance in Undirected Graphs

• Def: The distance between two vertices $u$ and $v$ in an undirected graph $G=(V,E)$ is
  • the minimum of the path lengths over all $u$-$v$ paths.
  • the depth of $u$ in $T_v$: a BFS tree from $v$
• We write it as $d(u,v)$. It satisfies the properties
  • $d(u,u) = 0$, for all $u \in V$
  • $d(u,v) = d(v,u)$, for all $u,v \in V$
  • $d(u,v) \leq d(u,w) + d(w,v)$, for all $u,v,w \in V$
Testing Connectedness

• How can we determine whether G is connected?

Pick a point u, show that all other vertices can be reached.
Level-Order Tree Traversal

```java
public static <E> void levelOrder(BinaryTree<E> root) {
    if (root.isEmpty()) return;

    // The queue holds nodes for in-order processing
    Queue<BinaryTree<E>> q = new QueueList<BinaryTree<E>>();
    q.enqueue(root); // put root of tree in queue

    while (!q.isEmpty()) {
        BinaryTree<E> next = q.dequeue();  //
        doSomething(next);
        if (!next.left.isEmpty()) q.enqueue(next.left());  //
        if (!next.right.isEmpty()) q.enqueue(next.right());  //
    }
}
```
Reachability: Breadth-First Search

BFS(G, v)  // Do a breadth-first search of G starting at v

```
Count ← 0
Create a queue Q
mark v as visited
Count++
enqueue v
Push S
while Q is not empty
  cur ← dequeue
  for each neighbor u of cur
    if u is not visited
      mark u as visited
      Count++
      enqueue u
      Push
return Count
// Compare Count to |V| in G. (If Count = |V| then G is connected)
```
BFS Reflections

• The BFS algorithm traced out a tree $T_v$: the edges connecting a visited vertex to (as yet) unvisited neighbors
• $T_v$ is called a *BFS tree of G with root v* (or *from v*)
• The vertices of $T_v$ are visited in *level-order*
• This reveals a natural measure of distance between vertices: the length of (any) shortest path between the vertices
DFS Reflections

• The DFS algorithm traced out a tree different from that produced by BFS
  • It still consists of the edges connecting a visited vertex to (as yet) unvisited neighbors
• It is called a DFS tree of $G$ with root $v$ (or from $v$)
• Vertices are visited in pre-order w.r.t. the tree
• By manipulating the stack differently, we could produce a post-order version of DFS
• And perhaps write DFS recursively…. 
Tree Traversals

public void preOrder(BinaryTree t) {
    if (t.isEmpty())
        return;
    doSomething(t);
    preOrder(t.left());
    preOrder(t.right());
}

+   7
*   2 3
Reachability: Depth-First Search (Recursive)

DFS(G, v)

Set \( v \) as visited

\text{count} \leftarrow 1

for each neighbor \( u \) of \( v \)

\begin{align*}
&\text{if } u \text{ is not visited} \\
&\quad \text{count} \leftarrow \text{DFS}(G, u)
\end{align*}

\text{return count}
Today’s Outline

• Graphs
  • Reachability
  • Graph Coloring
    • Lab10
Greedy Algorithms

• A **greedy algorithm** attempts to find a globally optimum solution to a problem by making locally optimum (greedy) choices

• Example: Walking in Manhattan

• Example: Graph Coloring
  • A *(proper) coloring* of a graph $G = (V, E)$ is an assignment of a value (color) to each vertex so that adjacent vertices get different values (colors)
  • Typically one strives to minimize the number of colors used
Graph Coloring Example

$\text{\textcolor{red}{c}} = \{a, d\}$
$\text{\textcolor{blue}{c}} = \{b, c\}$
$\text{\textcolor{green}{c}} = \{e\}$
$\text{\textcolor{purple}{c}} = \text{V}$

$\text{\textcolor{red}{c}} \cup \text{\textcolor{blue}{c}} \cup \text{\textcolor{green}{c}} \cup \text{\textcolor{purple}{c}} = \text{V}$
Here’s a greedy coloring algorithm for coloring $G$:

1. Build a collection $C = \{C_1, \ldots, C_k\}$ (set of set of vertices).
2. Set $i = 0$; $V =$ all vertices in $G$; $C_i = \{}$ // empty set
3. While $V$ has more vertices:
   - For each vertex $u$ in $V$:
     - If $u$ is not adjacent to any vertex of $C_i$:
       - Add $u$ to $C_i$.
     - Add $C_i$ to $C$.
     - Remove all vertices of $C_i$ from $V$.
   - Increment $i$.
4. Return $C$ as the coloring.
Today’s Outline

• Graphs
  • Reachability
  • Graph Coloring
• Lab10
Lab 10 : Exam Scheduling

Find a schedule (set of time slots) for exams so that

- No student has two exams in the same slot
- Every course is in a slot
- The number of slots is as small as possible

This is just the graph coloring problem in disguise!

- Each course is a vertex
- Two vertices are adjacent if the courses share students
- A slot must be an independent set of vertices (that is, a color class)
Lab 10 Notes: Using Graphs

- Create a new graph in structure5
  - GraphListDirected, GraphListUndirected,
  - GraphMatrixDirected, GraphMatrixUndirected

```
Graph <V, E> g = new GraphListUndirected<V, E>();
```
Lab 11: Useful Graph Methods

- void add(V label)
  - add vertex to graph
- void addEdge(V vtx1, V vtx2, E label)
  - add edge between vtx1 and vtx2
- Iterator<V> neighbors(V vtx1)
  - Get iterator for all neighbors to vtx1
- boolean isEmpty()
  - Returns true iff graph is empty
- Iterator<V> iterator()
  - Get vertex iterator
- V remove(V label)
  - Remove a vertex from the graph
- E removeEdge(V vLabel1, V vLabel2)
  - Remove an edge from graph