Today’s Outline

- Graphs
  - Reachability
  - Graph Coloring
    - Lab10

[TAP] Sum of degrees

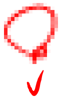
- Let *deg(v) be* the degree of a vertex v. Is the following statement true?
- For any graph *G = (V,E)*

$$\sum_{v \in V} \deg(v) = 2\,|E\,|$$

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

T

in dg $(v) = 1$
out dg $(v) = 1$



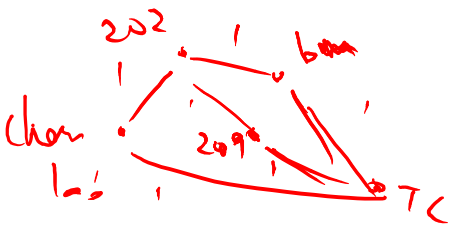
deg $(v) = 2$

Distance in Undirected Graphs

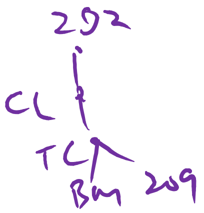
- Def: The *distance* between two vertices *u* and *v* in an undirected graph *G=(V,E)* is
  - the minimum of the path lengths over all *u*-*v* paths.
  - the depth of *u* in T_v: a BFS tree from *v*
- We write it as *d(u,v).* It satisfies the properties
  - $d(u,u) = 0$, for all $u \in V$
  - $d(u,v)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

aside

triangle inequality



uniform cost search



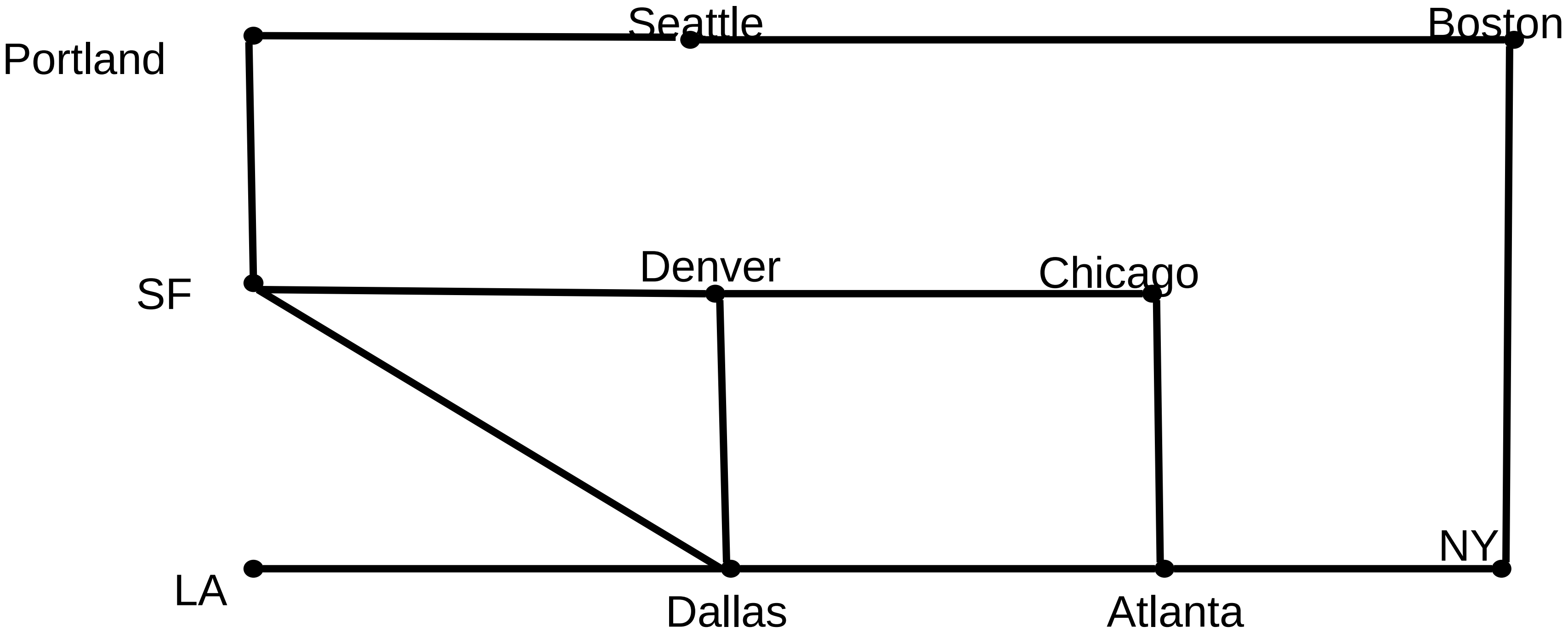
DFS tree

BFS tree

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

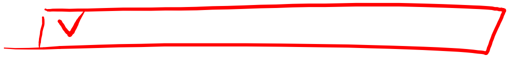
```
public static <E> void levelOrder(BinaryTree<E> root) {
   if (root.isEmpty()) return;

   // The queue holds nodes for in-order processing
   Queue<BinaryTree<E>> q = new QueueList<BinaryTree<E>>(); ①
   q.enqueue(root); // put root of tree in queue ②

   while(!q.isEmpty()) {
      BinaryTree<E> next = q.dequeue(); ③
      doSomething(next);
      if(!next.left().isEmpty())  q.enqueue(next.left()); ④
      if(!next.right().isEmpty()) q.enqueue(next.right());
   }
}
```

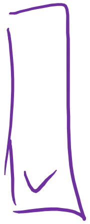
Reachability: Breadth-First Search

DFS

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

```
count ← 0
create a ~~queue Q~~ stack S
mark v as visited
count++
~~enqueue~~ push v
while ~~Q~~ S is not empty
    cur ← ~~dequeue~~ pop
    for each neighbor u of cur
        if u is not visited
            mark u as visited
            count++
            ~~enqueue~~ push u
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

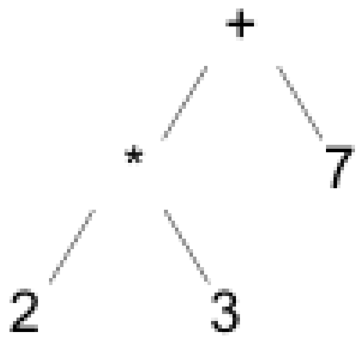
DFS Reflections

- 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

postOrder
inOrder

```
public void preOrder(BinaryTree t) {

    if (t.isEmpty())
        return;

    doSomething(t);

    preOrder(t.left());
    preOrder(t.right());

}
```



Reachability: Depth-First Search (Recursive)

DFS(G, v)

```
Set v as visited
count++
for each neighbor u of v
    if u is not visited
        count += DFS(G, u)

return count
```

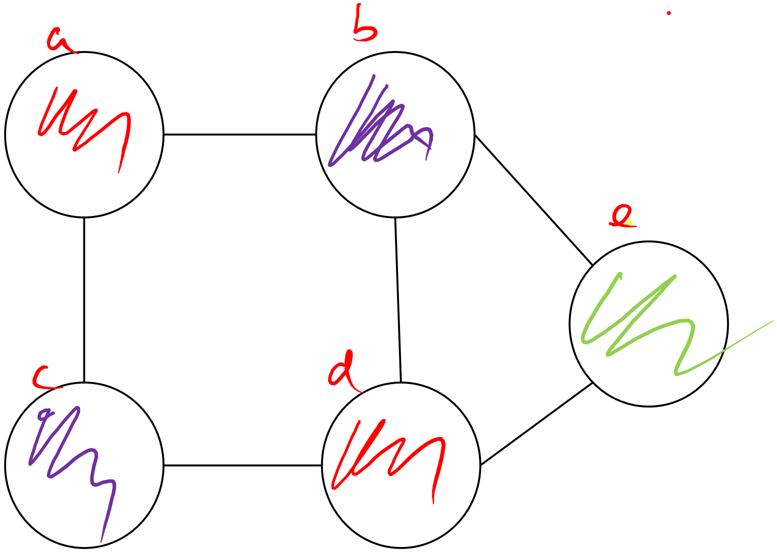
Today’s Outline

- 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



$C_0 = \{a, d\}$

$C_1 = \{b, c\}$

$C_2 = \{e\}$

$C_0 \cup C_1 \cup C_2 = V$

Greedy Coloring : Math

Here's a greedy coloring algorithm for coloring G

goal

```
Build a collection $C = \{C_1, \dots, C_k\}$ (set of set of vertices)
$i = 0$; V= all vertices in G; $C_i = \{\}$ // empty set
while $V$ is 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 $V$
    $i$++;
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