CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #21, Searching in Graphs

John Ridgway

April 16, 2015

1 Errata

Errors in Balancing a Binary Search Tree

- The code given in DJW to balance a binary search tree works well in the absence of duplicates.
- When given a tree with duplicates it does not work so well.
- I'm going to present a modified version that always works correctly; but first

Clicker Question #0

Given this code for adding to a BST, what will the height of adding the following elements, in order, be: 4, 1, 1, 1, 6, 5, 7? A. 1 B. 2 **C. 3** D. 4

```
public void add(T t) { root = addTo(t, root); }
private BSTNode<T> addTo(T t, BSTNode<T> node) {
  if (node == null) {
    return new BSTNode<T>(t, null, null); }
  if (t.compareTo(node.getData()) <= 0) {
    node.setLeft(addTo(t, node.getLeft())); }
  else {
    node.setRight(addTo(t, node.getRight())); }
  return node; }
```

The BST `balance` Method

```
public void balance() {
  T[] values = (T[]) new Comparable[size()];
  Iterator<T> iterator = inOrderIterator();
  int index = 0;
  while (iterator.hasNext()) {
    values[index] = iterator.next();
    index += 1;
  }
```

```
        root = bldTree(values, 0, index - 1);
      }
```

The BST `bldTree` Method

```
private BSTNode<T> bldTree(T[]vals,int l,int h)
{
  if (l == h) {
    return new BSTNode<T>(vals[l], null, null);
  } else if (l + 1 == h) {
    BSTNode<T> lNode = new BSTNode<T>(vals[l],
                                      null, null);
    return new BSTNode<T>(vals[h], lNode, null);
  } else {
    int mid = (l + h)/2;
    BSTNode<T> left = bldTree(vals,l,mid-1);
    BSTNode<T> right = bldTree(vals,mid+1,h);
    return new BSTNode<T>(vals[mid],left,right);
  }
}
```

2 Review

Review of `isPath` and DFS

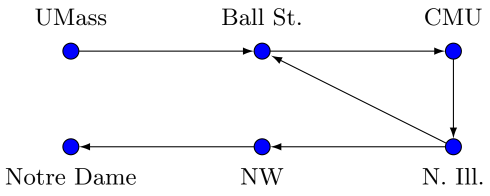
- Last time we discussed the "path problem," is there a path from vertex A to vertex B in the graph?
- We used depth-first-search implemented in a recursive manner to find paths.
- We needed to mark vertices when we saw them, otherwise we could get into loops.
- Today we'll look at an iterative version of DFS.

Iterative Depth-First Search

```
public boolean iterativeIsPath(
    GraphInterface<V> graph, V from, V to)
{
  Stack<V> stack = new Stack<V>();
  GraphMarker<V> marker = graph.getMarker();
  stack.push(from);
  do {
    V v = stack.pop();
    if (v == to) { return true; }
    if (! marker.isMarked(v)) {
      marker.mark(v);
      for (V neighbor : graph.getNeighbors(v)) {
        stack.push(neighbor); } }
  } while (! stack.isEmpty());
  return false;
}
```

A DFS Example

- For `isPath("CMU", "NW")`, push, pop, and mark `"CMU"`, then push `"N. Ill."`.
- Pop `"N. Ill."`, mark it, and push `"NW"` and `"Ball St."`.
- Pop `"Ball St."`, mark it, push `"CMU"`.
- Pop `"CMU"`, it's marked - ignore it.
- Pop `"NW"` and return `true`.



Clicker Question #1

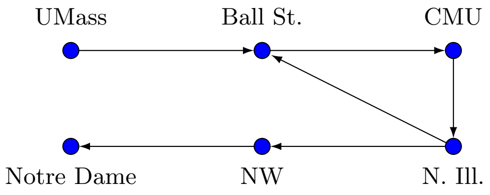
If we test `isPath("N. Ill.", "UMass")`, how many vertices will be marked when the method finishes?

A. 3

B. 4

C. **5**

D. 6



3 Weighted Graphs

Weighted Graphs

- We talked last lecture about putting weights on the edges of a graph. How would we represent that?
- It's a small modification to our `UnweightedGraphInterface` to get a `WeightedGraphInterface`.
- In fact the only things we need to do are to modify the `addEdge` method to take a weight as well as endpoints, and add a `getEdgeWeight` method to return the weight of an edge.

`WeightedGraphInterface`

```
import java.util.Iterator;
import java.util.List;
public interface WeightedGraphInterface<V>
{
  int getNumVertices();
  boolean isEmpty();
  void addVertex(V vertex);
  boolean hasVertex(V vertex);
  List<V> getVertices();
  Iterator<V> vertexIterator();
  void addEdge(V from, V to, double weight);
  double getEdgeWeight(V from, V to);
  boolean hasEdge(V from, V to);
  List<V> getNeighbors(V vertex);
  GraphMarker<V> getMarker();
}
```

The Implementation

Let's start with the `UnweightedDenseGraph` and figure out what we need to change:

- we need a way of storing the edge weights;
- the constructor will be different;
- we need a new `addEdge` method, since the signature is different;
- we need to write a `getEdgeWeight` method, since this does not exist in the other class; and
- it turns out that we need a new version of `hasEdge`.

```
import java.util.ArrayList;
import java.util.Arrays;
public class WeightedDenseGraph<V>
  implements WeightedGraphInterface<V>
{
  protected ArrayList<V> vertices;
  public static final double NULL_EDGE =
    Double.POSITIVE_INFINITY;
  private double[][] edges;

  public WeightedDenseGraph(int maxV) {
    super(maxV);
    edges = new double[maxV][maxV];
    for (double[] array : edges) {
      Arrays.fill(array, NULL_EDGE); } }
```

```
protected boolean hasEdge(int from, int to) {
  return ! Double.isInfinite(edges[from][to]); }

public void addEdge(V from, V to, double w) {
  int fromIndex = getIndexOf(from);
  int toIndex = getIndexOf(to);
  edges[fromIndex][toIndex] = w; }

public double getEdgeWeight(V from, V to) {
  return edges[getIndexOf(from)]
              [getIndexOf(to)]; }
```

4 Extracting the Commonalities

Extracting the Commonalities

- As we have seen, the `UnweightedGraphInterface` and `WeightedGraphInterface` interfaces are very similar; as are the two implementing classes.
- Actually, they are not implemented as shown earlier. Those slides actually drew lines from multiple files.
- In addition to the two interfaces there is a base interface `GraphInterface` which they extend, and a base class `BaseGraph` that the two implementing classes extend.

GraphInterface

```
import java.util.Iterator;
import java.util.List;

public interface GraphInterface<V>
{
  int getNumVertices();
  boolean isEmpty();
  void addVertex(V vertex);
  boolean hasVertex(V vertex);
  List<V> getVertices();
  Iterator<V> vertexIterator();
  boolean hasEdge(V from, V to);
  List<V> getNeighbors(V vertex);
  GraphMarker<V> getMarker();
}
```

The Other Interfaces

```
public interface UnweightedGraphInterface<V>
  extends GraphInterface<V>
{
  void addEdge(V fromVertex, V toVertex);
}
```

```
public interface WeightedGraphInterface<V>
  extends GraphInterface<V>
{
  void addEdge(V from, V to, double weight);
  double getEdgeWeight(V from, V to);
}
```

The Real UnweightedDenseGraph

```
public class UnweightedDenseGraph<V>
  extends BaseGraph<V>
  implements UnweightedGraphInterface<V> {
  private boolean[][] edges;

  public UnweightedDenseGraph(int maxV) {
    super(maxV);
    edges = new boolean[maxV][maxV]; }

  protected boolean hasEdge(int fromI, int toI) {
    return edges[fromI][toI]; }

  public void addEdge(V from, V to) {
    int fromIndex = getIndexOf(from);
    int toIndex = getIndexOf(to);
    edges[fromIndex][toIndex] = true; } }
```

The Real WeightedDenseGraph **Part I**

```
public class WeightedDenseGraph<V>
  extends BaseGraph<V>
  implements WeightedGraphInterface<V>
{
  public static final double NULL_EDGE =
    Double.POSITIVE_INFINITY;
  private double[][] edges;

  public WeightedDenseGraph(int maxV) {
    super(maxV);
    edges = new double[maxV][maxV];
    for (double[] array : edges) {
      Arrays.fill(array, NULL_EDGE); } }
```

The Real WeightedDenseGraph **Part II**

```
  protected boolean hasEdge(int from, int to) {
    return ! Double.isInfinite(edges[from][to]); }

  public void addEdge(V from, V to, double w) {
    int fromIndex = getIndexOf(from);
    int toIndex = getIndexOf(to);
    edges[fromIndex][toIndex] = w; }

  public double getEdgeWeight(V from, V to) {
    return edges[getIndexOf(from)]
                [getIndexOf(to)]; }
}
```

5 Breadth-first Search

Breadth-first Search

- What would happen if we took the iterative `isPath` method and replaced the stack with a queue?
- The answer: we would get a breadth-first search instead of a depth-first search.

`breadthFirstIsPath`

```
public boolean breadthFirstIsPath(
    GraphInterface<V> graph, V from, V to)
{
  Queue<V> queue = new LinkedList<V>();
  GraphMarker<V> marker = graph.getMarker();
  queue.add(from);
  do {
    V v = queue.remove();
    if (v == to) { return true; }
    if (! marker.isMarked(v)) {
      marker.mark(v);
      for (V neighbor : graph.getNeighbors(v)) {
        queue.add(neighbor); } }
  } while (! queue.isEmpty());
  return false;
}
```

Comparing DFS and BFS

- Both isPath methods will examine most or all of the edges in the graph in the worst case.
- If we use an array-based representation for the graph, with n vertices and e edges, we will spend $O(n^2)$ time making the vertex queues, but only $O(1)$ time per edge otherwise.
- The list-based representation will use only $O(e)$ total time overall, which is faster if the graph is sparse.
- BFS has the advantage that it will always find a path with the fewest number of edges.
- This is because it first puts all vertices one step from the start onto the queue, then all vertices two steps away, then all vertices three steps away, and so on until it finds the goal.
- Of course, if the edges are weighted, this might not be the shortest path by weight.
- BFS might find a path that is shorter than DFS would find.

- If the shortest path is long, BFS could spend a lot of time looking at all the shorter paths, while DFS *might* hit upon the correct path quickly.
- BFS will use more memory in many cases, as there are typically more than r nodes at distance r from the start.
- Obligatory XKCD reference: `http://xkcd.com/761`.

6 Implementing Graphs with Adjacency Lists

Adjacency Matrices

- The array-based implementation, called an **adjacency matrix**, is simple but on a graph with n vertices, it always takes $O(n^2)$ memory locations, one for each entry in the edge array.
- The adjacency matrix approach works well for highly-connected graphs with lots and lots of edges.
- Graphs in applications are usually **sparse**, meaning that each vertex has relatively few neighbors. A sparse graph of n vertices might have only $O(n)$ edges, and its matrix would be mostly `false`.

Adjacency Lists

- Another representation called an **adjacency list** saves space, and can save time if the algorithms are designed to use it efficiently.
- An adjacency list has an array of linked lists of vertices, one for each vertex. The list for vertex v has all of v's neighbors, so there is only one entry for each edge of a directed graph, or two for each edge of an undirected graph.
- We can implement the methods of the `WeightedGraph` interface fairly easily.
- For example, the list of neighbors just takes its entries from the list for the given vertex.
- In CMPSCI 311 most of the graph algorithms you'll see assume that the graph is given as an adjacency list. We measure the time complexity of these algorithms in terms of both n, the number of vertices, and e, the number of edges.

WeightedSparseGraph **Header**

```
import java.util.ArrayList;
import java.util.List;

public class WeightedSparseGraph<V>
  extends BaseGraph<V>
  implements WeightedGraphInterface<V> {
  private GraphNode[] edges;

  public WeightedSparseGraph(int maxVertices) {
    super(maxVertices);
    edges = new GraphNode[maxVertices];
  }

  protected boolean hasEdge(int fromI, int toI) {
    return findGraphNode(fromI, toI) != null;
  }

  protected GraphNode findGraphNode(V from, V to) {
    return findGraphNode(getIndexOf(from),
                         getIndexOf(to)); }

  protected GraphNode findGraphNode(int fromI,
                                    int toI) {
    for (GraphNode node = edges[fromI];
         node != null; node = node.getNext()) {
      if (node.getData()==toI) { return node; }
    }
    return null; }

  public void addEdge(V from, V to, double weight) {
    int fromI = getIndexOf(from);
    int toI = getIndexOf(to);
    edges[fromI] = new GraphNode(toI, weight,
                                 edges[fromI]); }

  public double getEdgeWeight(V from, V to) {
    GraphNode node = findGraphNode(from, to);
    if (node == null) {
      return Double.POSITIVE_INFINITY;
    } else {
      return node.getWeight();
    }
  }

  public List<V> getNeighbors(V from) {
    int fromI = getIndexOf(from);
    List<V> neighbors = new ArrayList<V>();
    for (GraphNode v = edges[fromI];
         v != null; v = v.getNext()) {
      neighbors.add(vertices.get(v.getData()));
    }
    return neighbors;
  }
}
```