CS211, Lecture 26
More Graphs

Announcements:

- L25 code fixed!
- makeup assignment ("A7") info
- grades

Overview:

- motivation for algorithms
- general search
- DFS revisted
- BFS revisted
- shortest path algorithm for unweighted graph

1. Motivation

1.1 Paths

- find ways to reach/find/collect/organize information from network of nodes
- focus of a lot of research!

1.2 Reachability

- is there a path from a given node to another node?
- ex) find the solved state of N-Puzzle from scrambled state

1.3 Minimal Path

- find the shortest path from a node to another
- find the shortest path from every node to another
- use weights to find min/max distances

1.4 Cycles

- ex) Traveling Salesman problem
- find the smallest length cucle that passes through all nodes
- no one knows if there is an efficient algorithm for this (NP/NP-complete problems)

2. Search

- kind of handy that we have explicit graph
- why?
  - verticies created ahead of time
  - stored in hash table
  - so, only need to look up a node…

```
public boolean search(Object o) {
    if (verticies.get(o)==null)
        return false;
    return true;
}
```


3. Traversal

3.1 Traversal

- like search for node, but now search for everything
- visit nodes
- also called walk
- see lists, trees, …

3.2 Why?

- want to find things
- want to way to something
- want to process everything

3.3 Types

- DFS (**`Digraph.java`**)
- BFS (**`Digraph.java`**)
- topological sort
- random
- more?

3.4 Applications

- test for cycles (DS&A)
- connectedness (DS&A)

3.5 DFS

- depth-first search
- process:
  - start with origin
  - visits a neighbor
  - visits neighbor of neighbor and so on…
  - stops when can't find unvisited neighbor
  - backs up to previous node and searches for new unvisited neighbor and so on…
- results in visiting all the nodes in a connected graph
- the DFS path never repeats a node
- show/print path as { v1, v2, … vn }:
  - left side (v1) is origin
  - each visited node inserted to right


```
public SeqStructure getDFS(Object origin) {

    resetVerticies();
    SeqStructure toDo = new StackAsList();
    SeqStructure path = new QueueAsList();

    Vertex originVertex = (Vertex)verticies.get(origin);
    originVertex.visit();
    toDo.put(originVertex);
    path.put(originVertex);

    while(!toDo.isEmpty()) {
        Vertex currentVertex = (Vertex)toDo.get();
        Iterator edges = currentVertex.getEdgeIterator();
        boolean found = false;
        Vertex nextVertex = null;
        while (!found && edges.hasNext()) {
            Edge currentEdge = (Edge) edges.next();
            Vertex trialVertex = currentEdge.getDest();
            if(!trialVertex.isVisited()) {
                found = true;
                nextVertex = trialVertex;
            }
        }

        if (nextVertex != null) {
            toDo.put(currentVertex);
            nextVertex.visit();
            toDo.put(nextVertex);
            path.put(nextVertex);
        }
    }
    return path;
}
```


3.6 BFS

- breadth-first search (called level-order in trees)
- process:
  - visit origin (record this node)
  - visit each of origin's neighbors (record each node in order visited)
  - visit neighbors of each neighbor (record those nodes) and so forth
- show/print path as { v1, v2, … vn }:
  - left side (v1) is origin
  - each visited "level" is inserted to right of previous node
  - so might wish to think as { {level 1}, {level 2}, … }


```
public SeqStructure getBFS(Object origin) {

    resetVerticies();
    SeqStructure toDo = new QueueAsList();
    SeqStructure path = new QueueAsList();

    Vertex originVertex = (Vertex)verticies.get(origin);
    originVertex.visit();
    toDo.put(originVertex);
    path.put(originVertex);

    while(!toDo.isEmpty()) {
        Vertex currentVertex = (Vertex)toDo.get();

        for (Iterator edges =
                              currentVertex.getEdgeIterator();
                              edges.hasNext(); ) {

            Edge currentEdge = (Edge) edges.next();
            Vertex nextVertex = currentEdge.getDest();

            if(!nextVertex.isVisited()) {
                nextVertex.visit();
                toDo.put(nextVertex);
                path.put(nextVertex);
            }
        }

    }

    return path;

}
```

3.7 Example

- see **`TestDigraph.java`**
- contains more examples

```
    Digraph g = new Digraph();
    g.addVertex("A");
    g.addVertex("B");
    g.addVertex("C");
    g.addVertex("D");
    g.addVertex("E");
    g.addVertex("F");
    g.addVertex("G");
    g.addEdge("A","B");
    g.addEdge("A","C");
    g.addEdge("B","D");
    g.addEdge("B","E");
    g.addEdge("C","F");
    g.addEdge("C","G");

    System.out.println(g.getBFS("A"));
    System.out.println(g.getDFS("A"));

/*
     BFS: [A B C D E F G]
     DFS: [A B D E C F G]
 */
```


4. Shortest Path Algorithms

4.1 Assumptions

- edge-weighted graph (unitary or more)
- weight is a cost of using an edge
- graphs may be directed or undirected
- non-negative edge weights!

4.2 Why?

- want to find best/cheapest/least effort between points
- travelling is classic example

4.3 Terms

- ***weighted path length*** = sum of weights on path
- ***unweighted path length*** = sum of paths (weights = 1)

4.4 SSSP

- to find shortest path from A to B, need to find *shortest path from A to all other nodes*
  - why? another node might provide a better path
  - see DS&A 16.4.1: basically, knowing all the path lengths might mean you can find a shorter route
- this problem called ***single-source shortest path*** problem

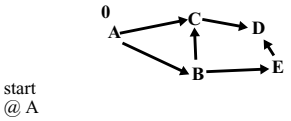
5. SSSP for Unweighted Graphs

5.1 The Gist

- based on BFS
- all traverse trees, keep track of increasing path lengths to a particular node
- algorithm finds only 1 path if multiple paths are smallest and have same value
- will need to modify classes again
  - need to find all paths to end node
  - need to keep track of length to current node (so can have length values after traversal)
  - so, need to keep track of previous node

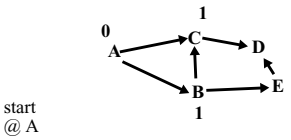
5.2 Process for SSSP

- finding shortest path from A to B means counting edges
- smallest number of edges gives shortest path
- since starting at A, start counting @ A:



start
@ A

- try to find final node, so count edges while looking:



start
@ A

- which will get the smallest node cost if check when reach final node and backtrack

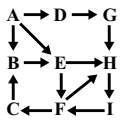
5.3 Path Cost

- for path to node v, distance to v is Dv
- for v → w, Dw = Dv + 1
- helpful convention: default node cost for SSSP: Dw = ∞
- we're not really using the Dw convention, though many implementations do (we have **`visited`**, which is our way of tagging visited nodes)

5.4 Algorithm (BFS for Destination)

- start with origin
- put origin in Q
- not done, so take first node from Q
- find edges from node
- for each node, if it's not already been visited
  - tag it, set cost, set prev node, put in Q
  - if the node is dest, stop processing!

5.5 Example: Part 1–Build Q




```
origin: A; end: I
get A: Q = [A]
get A: Q = [ ]
     process (AD),(AE),(AB) (nodes D,E,B)
     for each node, put in Q [D,E,B], tag, set cost (1),
     set prev (A), check if dest (no for all)
get D: Q = [E,B]
     process (DG) (node G)
     tag G, put in Q [E,B,G], set cost (G=2),
     set prev (D), check if dest (no)
get E: [B,G]
     process (EH), (EF) (nodes H,F)
     tag nodes, put in Q [B,G,H,F], set costs (H=F=2)
     & prev (E), check if dest (no for all)
get B: [G,H,F]
     nothing new -- no processing
get G: [H,F]
     process (GH)
     nothing new (H already visited)
get H: [
     tag I, put in Q [H,F,I], set cost (3), set prev (H)
     check if dest -- yes! STOP!
```

5.6 Example: Part 2–Build Stack

```
put last vertex (end) into Stack: [I]
set last vertex to prev of last index: H
put last vertex into Stack: [H,I]
set last vertex to prev of last index: E
put last vertex into Stack: [E,H,I]
set last vertex to prev of last index: A
put last vertex into Stack: [A,E,H,I]
no more prev (prev is null)
return Stack, which contains shortest path
```


5.7 Code

```
public SeqStructure unweightedShortestPath(Object origin,
                                           Object end) {
    resetVerticies();
    boolean done = false;
    SeqStructure toDo = new QueueAsList();
    SeqStructure path = new StackAsList();

    Vertex originVertex = (Vertex)verticies.get(origin);
    Vertex endVertex = (Vertex)verticies.get(end);
    originVertex.visit();
    toDo.put(originVertex);

    while(!done && !toDo.isEmpty()) {
        Vertex currentVertex = (Vertex)toDo.get();

        for (Iterator edges=currentVertex.getEdgeIterator();
              !done && edges.hasNext(); ) {

            Edge currentEdge = (Edge) edges.next();
            Vertex nextVertex = currentEdge.getDest();

            if(!nextVertex.isVisited()) {
                nextVertex.visit();
                nextVertex.setCost(1+
                                   currentVertex.getCost());
                nextVertex.setPrev(currentVertex);
                toDo.put(nextVertex);
            }

            if (nextVertex.equals(endVertex))
                done = true;

        } // end for

    } // end while

    path.put(endVertex);
    while(endVertex.hasPrev()) {
        endVertex = endVertex.getPrev();
        path.put(endVertex);
    }
    return path;
}
```


6. Exercises

- Use recursion to rewrite (and simplify) the DFS code. You might need a helper method.
- Write a program that finds all DFS/BFS paths in a graph. Is this problem related to an implicit graph search
- Rewrite the shortest path algorithm such that it uses the convention of "infinite" costs as the check for stopping instead of tagging of nodes.
- Try to figure out an algorithm for finding the shortest path when there are edge weights.