

CS231 Fall 2013 Lecture 33

Stephanie R Taylor

November 20, 2013

1 Administrative Topics

- Thanks for coming to the celebration yesterday! Was it helpful?

2 Graph problems

Now, what do people do with graphs?

1. Is there a path that uses each edge exactly once? [Eulerian Path Problem]
2. Is there a path that goes through each node exactly once? [Hamilton Path Problem]
3. Is there a circuit that uses each edge exactly once? [Eulerian circuit Problem]
4. Is there a circuit that goes through each node exactly once? [Hamilton circuit Problem]
5. Does it have particular data in it? [a search problem]
6. Is it connected?
7. Are there any cycles?

8. Is there a path between two given nodes? If so, what is it?
9. What is the shortest path (least number of edges), if any, between two given nodes? [Shortest Path Problem]
10. What is the shortest path (sum of weights of edges) between two given nodes?
11. What are the lengths of the shortest paths between all pairs of nodes? [All-pairs Shortest Path Problem]
12. What is the shortest tour of all nodes in a complete graph (where shortest means the least sum of the weights of edges)? [Travelling Salesperson Problem]
13. Can the nodes be colored red and blue so that red nodes have edges only to blue nodes and vice versa? [Bipartite Graph Problem]
14. What is the fewest number of colors needed to color the nodes so that adjacent nodes have different colors? [Chromatic Number of a graph]
15. Can the graph be drawn in the plane so that no edges cross? [Planarity Problem]
16. How many paths of a given length are there between two given nodes?
17. Can each node be represented by an interval on the real line so that adjacent nodes correspond to overlapping intervals? [Interval Graph Problem]

3 Traversing a graph

In many of the problems, we need to use some graph traversal algorithm where we are visiting all nodes. For example, all three problems for your project involve traversals. All these traversals can be done recursively or with stacks or queues. During the traversal, there are three categories of nodes: Nodes already visited, nodes that are unvisited neighbors of visited nodes, and the remaining nodes. We will mark the nodes when visited and we will keep a collection C of unvisited neighbors of visited nodes.

General graph traversal algorithm:

```
unmark all nodes
create a new empty collection C of nodes
choose a starting node and add it to C
while C is not empty do:
    remove a node n from C
    mark n as visited
    for all neighbors p of n:
        if p is unmarked and not already in C:
            add p to C
```

Determining if a graph is connected

How might we determine connectedness? Do a traversal and then check whether all nodes have been visited.

```
// returns true if the graph is connected
public boolean isConnected() {
    unmark all nodes
    create a new empty collection C of nodes
    choose a starting node and add it to C
    while C is not empty do:
        remove a node n from C
        mark n as visited
        for all neighbors p of n:
            if p is unmarked and not already in C:
                add p to C
    if there are still unvisited nodes:
        return false
    else:
        return true
}
```

Determining if a graph has any cycles

To see if a graph has cycles, what can we do? Do the general graph traversal algorithm except if neighbor p is marked or already in C, then return true.

```
// returns true if the graph has cycles
public boolean hasCycles() {
    unmark all nodes
    create a new empty collection C of nodes to be visited
    choose a starting node and add it to C
    while C is not empty do:
        remove a node n from C
        mark n as visited
```

```

        for all neighbors p of n:
            if p is unmarked and not already in C:
                add p to C
            else:
                return true
    return false
}

```

This doesn't quite work. Why not? [A graph with 2 connected nodes will be called cyclic.] For undirected graphs with no loops or multiple edges, we only consider cycles of length 3 or more. How do we handle this situation? [For each node, keep track of its parent and don't count that node as a neighbor when looking for cycles.]

```

// returns true if the graph has cycles
public boolean hasCycles() {
    unmark all nodes
    create a new empty collection C of nodes to be visited
    choose a starting node n
    set n's parent to null
    add n to C
    while C is not empty do:
        remove a node n from C
        mark n as visited
        for all neighbors p of n other than p's parent:
            if p is unmarked and not already in C:
                set p's parent to n
                add p to C
            else:
                return true
    return false
}

```

Alternatively, the only way to get a cycle is for one node, when visited to have two neighbors that are already visited or in C. This is easier to code.

```

// returns true if the graph has cycles
public boolean hasCycles() {
    unmark all nodes
    create a new empty collection C of nodes to be visited
    choose a starting node n and add n to C
    while C is not empty do:
        remove a node n from C
        mark n as visited
        for all neighbors p of n:

```

```

        if p is already in C:
            return true
        else if p is unmarked:
            add p to C
    return false
}

```

However, neither of these algorithms work if the graph is disconnected. In that case, you need to keep checking each connected component:

```

// returns true if the graph has cycles
public boolean hasCycles() {
    unmark all nodes
    create a new empty collection C of nodes to be visited
    while there are unmarked nodes
        choose an unmarked starting node n and add n to C
        while C is not empty do:
            remove a node n from C
            mark n as visited
            if n has two neighbors that are marked or in C:
                return true
        for all neighbors p of n:
            if p is unmarked and not already in C:
                add p to C
    return false
}

```

Determining if a graph is bipartite (two-colorable)

To see if a graph can be 2-colored, just pick a start node and color it blue. Then color all its neighbors red (if any of those neighbors are already blue, return false). Now, repeat. Every time a node is colored one color, try to color its neighbors the opposite color. If any of them are already colored the same color, return false. When you are done, if you haven't yet returned false, then return true.

```

// returns true if the graph is bipartite
public boolean isTwoColorable() {
    uncolor all nodes
    create a new empty collection C of nodes
    while there are uncolored nodes
        choose an uncolored starting node n
        color n blue and add n to C
        while C is not empty do:
            remove a node n from C

```

```

        for all neighbors p of n:
            if p is uncolored:
                color p the opposite color of n
                add p to C
            else if p is colored the same color as n:
                return false
    return true
}

```

Now what about this collection C that is used? Does it make any difference what kind of collection it is? Usually either a Queue or a Stack is used. What is the difference?

In the case of the stack, the next node to be popped off and visited is the one that was pushed on last, namely a neighbor of the most recently visited node that had unvisited neighbors. So you go from visited node to neighbor and then to a neighbor of that neighbor. When you reach a dead end, the next node to visit is a neighbor of an earlier visited node. You are traversing a tree but going in a depth-first-search fashion. [Do an example on the board]

If you recursively visit a node and then recursively visit its neighbors, you are also doing a DFS traversal.

What about a Queue? In that case, you visit next the node that was added to the queue first. That is, all neighbors of the starting node are visited first. Then all neighbors of those neighbors are visited. So the nodes of distance one from the start are visited first and then nodes of distance 2, etc. This is called breadth-first-search.