Week 3 - Friday

COMP 4500

Last time

- What did we talk about last time?
- Trees
- Graph connectivity
- Breadth-first search
- Representing graphs

Questions?

Depth First Search

Queues and stacks

- A **queue** is a set where we extract elements in **first-in, first-out** (FIFO) order
- A **stack** is a set where we extract elements in **last-in, first-out** (LIFO) order
- Both data structures can be efficiently implemented by a doubly-linked list

Implementing BFS

- BFS(***s***):
  - Set Discovered[***s***] = true
  - Set Discovered[***v***] = false for all ***v*** ≠ ***s***
  - Create list ***L***[0] and put ***s*** in it
  - Set layer counter ***i*** = 0
  - Set current BFS tree ***T*** = ∅
  - While ***L***[***i***] is not empty
    - Create list ***L***[***i*** + 1]
    - For each node ***u*** ∈ ***L***[***i***]
      - Consider each edge (***u***, ***v***) incident to ***u***
      - If Discovered[***v***] = false then
        - Set Discovered[***v***] = true
        - Add edge (***u***, ***v***) to the tree ***T***
        - Add ***v*** to list ***L***[***i*** + 1]
    - Increment layer counter ***i***

Running time for BFS

- Weak bound: $O(n^2)$
- We have at most n lists $L[i]$, taking $O(n)$ time to set up
- Each node occurs at most once in any list, so the total iterations of the For loops in the While is n
- When we consider a node u, it has at most n edges, each of which can be processed in constant time
- n iterations of the For loop taking at most $O(n)$ time each gives $O(n^2)$ time

Better running time for BFS

- Stronger bound: $O(n + m)$
- The argument is the same except that there might be fewer than n^2 edges
- The total number of edges considered (in an adjacency list representation) for all nodes u is $2m$ (because each edge is seen twice)
- Total time then is $O(n)$ set-up, $O(n)$ nodes checked, and $O(m)$ edges checked, giving $O(n + m)$ time

Implementing DFS (non-recursively)

- DFS(***s***):
  - Create stack ***S*** and put ***s*** in it
  - While ***S*** is not empty
    - Take a node ***u*** from ***S***
    - If Explored[***u***] = false then
      - Set Explored[***u***] = true
      - For each edge (***u***, ***v***) incident to ***u***
        - Add ***v*** to the stack ***S***

Notes about DFS

- Because the stack is FILO, we will actually process the adjacency list in reverse order from the recursive version of DFS
- We can find the DFS tree by adding a parent array to the algorithm
  - When we add node ***v*** to the stack because of edge (***u***, ***v***), we set parent[***v***] = ***u***
  - When we mark any node ***u*** (other than ***s***) as Explored, we add edge (***u***, parent[***u***]) to the DFS tree

Running time for DFS

- Again, we can get $O(n + m)$ time
- Adding and removing from the stack can be done in constant time
- A node will get added to the stack every time one of its adjacent nodes is explored
- The total number of nodes added (and removed) is bounded by the total degree of the graph, $2m$
- Running time is then $O(n + m)$

Finding all connected components

- Using either BFS or DFS, you will only find the connected component containing ***s***
- We can keep running either until we find all connected components
- The running time O(***n*** + ***m***) is actually based only on the nodes and edges in a particular connected component
- Finding all connected components will still only take O(***n*** + ***m***) for the whole graph

Three-Sentence Summary of Testing for Bipartiteness, Directed Connectivity, and Topological Sort

Testing for Bipartiteness

Bipartiteness

- Recall the definition of a bipartite graph:
  - A graph that can be partitioned into sets ***X*** and ***Y*** such that every edge has one end in ***X*** and the other in ***Y***
  - Or, you can think of nodes in set ***X*** as red and nodes in set ***Y*** as blue
- An alternative, equivalent definition of a bipartite graph is one that has no odd cycles

Detecting bipartiteness

- Pick a node and color it blue
- Color all of its neighbors red
- Keep going, coloring neighbors, alternating which color you use
  - Don't change the color of a node if it's already colored
- If there are any edges that start and end in the same color, it's not bipartite
- This algorithm is essentially BFS where, when adding a node to layer $L[i + 1]$, we color it red when $i + 1$ is even and blue when $i + 1$ is odd

Layer lemma

- Let T be a breadth-first search tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge of G.
- Then i and j differ by at most 1.
- Proof by contradiction:
  - Suppose i and j differ by more than one. Assume that $i < j - 1$. Since x is in layer L_i, the only nodes discovered from x belong to layers L_{i+1} and earlier. If y is a neighbor of x, it should have been discovered and put in layer L_{i+1} or earlier.

Claim of correctness

- Let ***G*** be a connected graph, and let L_1, L_2, … be the layers produced by BFS starting at node ***s***. Exactly one of the following two things must hold:

1. There is no edge of ***G*** joining two nodes of the same layer. In this case ***G*** is a bipartite graph in which the nodes in even-numbered layers can be colored red, and the nodes in odd-numbered layers can be colored blue.
2. There is an edge of ***G*** joining two nodes of the same layer. In this case, ***G*** contains an odd-length cycle, and so it cannot be bipartite.

Proof of correctness

- Case 1. No edges join two nodes in the same layer. By the Layer Lemma, every edge of ***G*** joins nodes either in the same layer or in adjacent layers. Since no edges join nodes in the same layer, they always join nodes in adjacent layers, with different colorings. Thus, ***G*** is bipartite.

Proof of correctness (continued)

- Case 2: At least one edge joins two nodes of the same layer, x and y. Let x and y be in layer L_j. Let z be the node with the highest layer number possible while still being an ancestor of x and y in the BFS tree. Let z be in layer L_i, where $i < j$. There is a cycle in G from z down to x, from x to y, and then from y back to z. The length of the cycle is $(j - i) + 1 + (j - i) = 2(j - i) + 1$, which is odd. Thus, the graph is not bipartite.

Directed Connectivity

Directed graph representations

- It can be useful to extend the adjacency list representation for directed graphs
- As before, for node ***u***, we have a list of nodes that ***u*** connects to
- But we add a second list of nodes that connect to ***u*** as well
- In this way, we can efficiently determine all nodes that can reach ***u***

Directed DFS and BFS

- We can run DFS or BFS on a directed graph starting at ***s***
- Instead of getting a connected component, we will get a tree of nodes reachable from ***s***
  - Not all nodes will necessarily have a path back to ***s***
- We can also run DFS on reversed edges, yielding the tree of nodes that can reach ***s***

Strong connectivity

- A directed graph is **strongly connected** if, for all nodes ***u*** and ***v***, there is a path from ***u*** to ***v*** and a path from ***v*** to ***u***
- Nodes ***u*** and ***v*** are mutually reachable if you can reach ***u*** from ***v*** and ***v*** from ***u***
- If ***u*** and ***v*** are mutually reachable and ***v*** and ***w*** are mutually reachable, then ***u*** and ***w*** are mutually reachable

Strong components

- To see if a graph is strongly connected, pick a node ***s*** and run BFS from it
- Then run BFS on the reversed edge graph
- If both searches visit every node, it's strongly connected
- A **strong component** containing ***s*** is the set of all nodes ***v*** such that ***s*** and ***v*** are mutually reachable
- For any two nodes ***s*** and ***t*** in a directed graph, their strong components are either identical or disjoint

Topological Sort

Directed acyclic graph

- A directed acyclic graph (DAG) is a directed graph without cycles in it
- These can be used to represent dependencies between tasks
- An edge flows from the task that must be completed first to a task that must come after
- A cycle in such a graph would mean there was a circular dependency
- By running topological sort, we discover if a directed graph has a cycle, as a side benefit

Topological sort

- A **topological sort** gives an ordering of the tasks such that all tasks are completed in dependency ordering
- In other words, no task is attempted before its prerequisite tasks have been done
- There are usually multiple legal topological sorts for a given DAG

Quiz

Upcoming

Next time…

- Finish topological sort
- Greedy algorithms
- Scheduling

Reminders

- **Work on Assignment 2**
  - **Due next Friday before midnight**
- Read sections 4.1 and 4.2