CS 4349 Lecture—October 11th, 2017

Main topics for #lecture include #graph_traversal, #depth-first_search, and #DAGs.

Prelude

- Homework 5 due today.
- Homework 6 due Wednesday, October 18th.

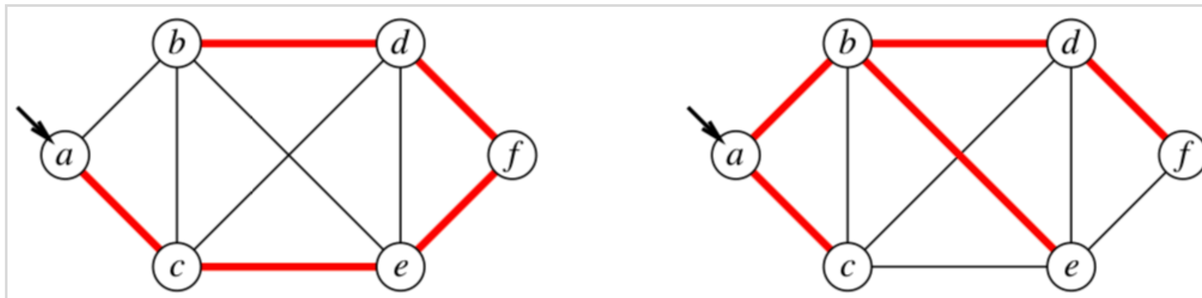
Graph Traversals

- Last time, we discussed a generic traversal algorithm that stores a pair of vertices in the bag. Using this algorithm, we can remember when visiting a vertex for the first time where we just came from. We call where we came from the *parent*.

```
TRAVERSE(s):
  put (∅,s) in bag
  while the bag is not empty
      take (p,v) from the bag              (⋆)
      if v is unmarked
          mark v
          parent(v) ← p
          for each edge vw                 (†)
              put (v,w) into the bag      (⋆⋆)
```

- Lemma: Traverse marks each vertex of a connected graph exactly once, and the set of pairs (p, parent(p)) with parent(p) ≠ emptyset form a spanning tree of the graph.
- The exact running time depends upon what kind of bag we use, but we do know some things.
  - Each vertex is marked once, so the loop (cross) runs exactly V times.
  - Each edge uv is added to the bag twice, once as (u, v) and once as (v, u), so * * runs 2E times.
  - And we can't take more out of the bag than we put in, so * is run 2E + 1 times.
- But what are some different bags we could use?
- We'll assume the graph is stored as an adjacency list, so the loop (cross) takes constant time per edge of v.
- Example 1: The bag is a stack.
  - This is the original depth-first search algorithm.
  - Each of * and * * takes constant time, so the whole algorithm takes O(E) time.
  - The spanning tree of parent edges is called a *depth-first spanning tree*.
  - The exact tree you get depends upon the starting vertex s and what order you visit

edges, but they tend to be long and skinny.
- We'll discuss them more in a little bit.



- Example 2: The bag is a queue.
  - Each * and * * takes constant time, so the whole algorithms takes O(E) time again.
  - Now the spanning tree of parent edges is called a *breadth-first spanning tree*.
  - The tree contains the shortest paths from s to other vertices (assuming each edge is the same length). You can find the distance between s and other vertices by just following the paths back up the spanning tree and counting edges.
  - Again, the shape varies depending upon s and the order you scan edges, but these trees tend to be short and bushy.
- Example 3: Edges have real number weights and the bag is a priority queue where we extract minimum weight edges.
  - We can call this a *shortest-first search*.
  - Each of * and * * takes O(log E) time so the search takes O(E log E) time overall.
  - This is actually an algorithm to compute a *minimum spanning tree*. A spanning tree that has minimum total edge weight.
  - Surprisingly, if the edge weights are distinct, then the minimum spanning tree is unique.
- If we used an adjacency matrix instead, then loop (cross) always takes O(V) time since we have to check which edges are present. Depth-first and breadth-first search require O(V^2) time. Shortest-first search takes O(V^2 + E log E) time.
- Finally, what if the graph is disconnected? We can use a wrapper around our Traverse algorithm.

```
TRAVERSEALL( ):
  for all vertices v
      if v is unmarked
          TRAVERSE(v)
```

- Since Traverse computes a spanning tree of each component, TraverseAll computes a spanning forest of the whole graph.

Depth-First Search

- So today, I want to focus on some of the properties and applications of one of these traversal algorithms. Depth-first search. Since it uses a stack, it's actually a bit easier just to work with the recursive algorithm directly.

```
DFS(v):
  if v is unmarked
      mark v
      for each edge vw
          DFS(w)
```

- However, it's useful to make two modifications.
- The first is to only recurse on unmarked vertices. This speeds up the algorithm in practice.
- The other is to add parent points and these two routines PreVisit and PostVisit which we can fill in later. These will be useful for doing other things later in the lecture.

```
DFS(v):
  mark v
  PREVISIT(v)
  for each edge vw
      if w is unmarked
          parent(w) ← v
          DFS(w)
  POSTVISIT(v)
```

- This idea of subtly modifying algorithms is going to be a running trend for this lecture.
- Like we saw, this algorithm will eventually mark and compute a spanning tree for any connected graph. But we get two additional properties since we're doing a depth-first search.
- Let T be a depth-first spanning tree of a connected undirected graph G computed by DFS(s).
- Lemma: For any node v, the vertices marked while executing DFS(v) are the proper dependents of v in T.
- Proof: T is also the recursion tree for DFS(s).
- For every edge vw in G, either v is an ancestor of w in T, or v is a descendant of w in T.
- Proof: Suppose v is marked before w. Then w is unmarked when DFS(v) is envoked but marked when DFS(v) returns, so the earlier lemma implies w is a proper descendent of v in T.
- The second lemma implies we can use T to divide the edges into two classes. *Tree* edges lie in T. *back* edges connect a node of T to one of its ancestors.

Application: Counting and Labeling Components

- Earlier, I discussed wrapping the traversal algorithm so you can visit every component. We can do this for our recursive DFS algorithm and include a PreProcess routine to help with

PreVisit PostVisit.

```
DFSALL(G):
    PREPROCESS(G)
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            DFS(v)
```

- By filling in the PreProcess, PreVisit, and PostVisit routines, we can write a simple algorithm for counting components and labeling vertices with their component.

```
COUNTANDLABEL(G):
    count ← 0
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            count ← count + 1
            LABELCOMPONENT(v, count)
    return count
```

```
LABELCOMPONENT(v, count):
    mark v
    comp(v) ← count
    for each edge vw
        if w is unmarked
            LABELCOMPONENT(w, count)
```

- PreProcess became setting the count to 0. PreVisit became setting comp(v) and PostVisit became nothing.
- Nothing special about DFS was used here. We could have changed the generic traversal algorithm instead.

Application: Preorder and Postorder Labeling

- Like with trees, we can label vertices by the order in which we visit them during a depth-first search.

```
PREPOSTLABEL(G):
    for all vertices v
        unmark v
    clock ← 0
    for all vertices v
        if v is unmarked
            clock ← LABELCOMPONENT(v, clock)
```

```
LABELCOMPONENT(v, clock):
    mark v
    pre(v) ← clock
    clock ← clock + 1
    for each edge vw
        if w is unmarked
            clock ← LABELCOMPONENT(w, clock)
    post(v) ← clock
    clock ← clock + 1
    return clock
```

- Or more simply, we could have just defined those three functions we used earlier by using a global clock variable.

```
PREPROCESS(G):
    clock ← 0
```

```
PREVISIT(v):
    pre(v) ← clock
    clock ← clock + 1
```

```
POSTVISIT(v):
    post(v) ← clock
    clock ← clock + 1
```

- Depth-first search has a nice property that the pre and post-visits for vertices are nested. In

other words, intervals [pre(u), post(u)] and [pre(v), post(v)] are either disjoint or one contains the other. In fact, if uv is an edge, then the intervals **must** be nested.

- Why? Suppose u is marked before v. Then pre(u) < pre(v). If v is a descendent of u, then post(u) > post(v) and otherwise, pre(v) > post(u).

Directed Graphs and DAGs (an Application)

- For the rest of the lecture, we'll consider directed graphs. The only change we need to make to DFS(v) is to loop over directed edges out of v.

```
DFSALL(G):
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            DFS(v)

DFS(v):
    mark v
    PREVISIT(v)
    for each edge v→w
        if w is unmarked
            DFS(w)
    POSTVISIT(v)
```

- One thing to note is we cannot simply count connected components like we did before. **draw a path** Depending on what order we loop over the vertices in DFSAll, we'll count somewhere between 1 and n components. *
- But we can decide if vertex u can *reach* vertex v, meaning there is a directed path from u to v. Just unmark all vertices and then call DFS(u).
- DFS is especially useful for working with directed acyclic graphs or DAGs.
- Some definitions: As I said Monday, a DAG has no directed cycles. A *source* vertex in a DAG has no incoming edges. A *sink* vertex in a DAG has no outgoing edges.
- We can check if a directed graph is a DAG in O(V + E) time by modifying our depth-first search. We'll use two main ideas. One is to add an vertex s to our graph connected to everything else, so that everything is reachable from s. The other is to extend our marks to include three different statuses, NEW, ACTIVE, or DONE. We'll mark a vertex ACTIVE when we first reach it. If we reach it again while it is ACTIVE, then we don't have a DAG.

```
ISACYCLIC(G):
    add vertex s
    for all vertices v ≠ s
        add edge s→v
        status(v) ← NEW
    return ISACYCLICDFS(s)

ISACYCLICDFS(v):
    status(v) ← ACTIVE
    for each edge v→w
        if status(w) = ACTIVE
            return FALSE
        else if status(w) = NEW
            if ISACYCLICDFS(w) = FALSE
                return FALSE
    status(v) ← DONE
    return TRUE
```

- We need to prove that the algorithm returns FALSE if and only if the graph has a directed cycle.
- Suppose we return FALSE. We found an edge v ➔ w where w is ACTIVE. But, ACTIVE vertices are on the recursion stack, meaning v is reachable from w. So there's a cycle from

w to v to w again.

- Suppose G has a directed cycle. Let w be the first vertex we visit on the cycle, and let v ➡ w be an edge on the cycle. Since v is reachable from w, we must call IsAcyclicDFS(v) during the execution of IsAcyclicDFS(w) unless we find some other cycle and return FALSE first. When we execute IsAcyclicDFS(v), we consider edge v ➡ w, and discover status(w) = ACTIVE. Then we return FALSE and it bubbles up through the recursive calls.