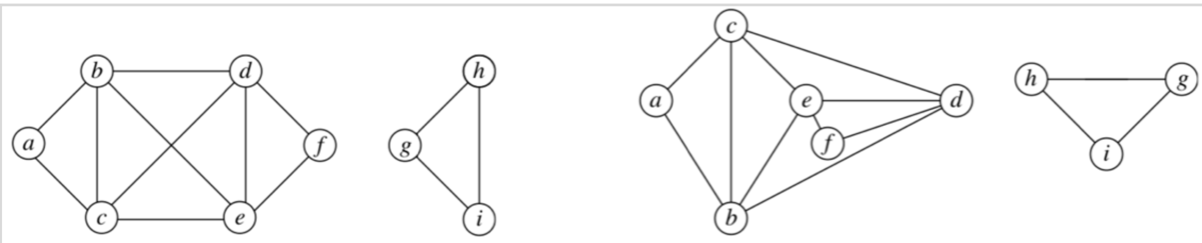
CS 6363.005.19S Lecture 12—February 28, 2019

Main topics are #graph_basics, including #breadth-first_search and #depth-first_search.

Prelude

- Homework 3 will be released today or tomorrow. It's going to be due **Thursday** March 14.
- Part of my duties as a professor include attending workshops and proposal review panels. Unfortunately, that means I won't be here for some of our scheduled meeting times.
- Thursday the 14th, your TA Jiashuai Lu will do a guest lecture on matroids, a family of structures that include the minimum spanning trees that I plan to cover next week. These provide a good target for greedy algorithms. I may ask about them in a homework, but since it's a guest lecture, I promise not to include anything about matroids in particular in any exams including the QE.
- Finally, I'll be gone during the last week of the semester. At our current pace, we'll have gone over everything that may appear on the QE by Tuesday of the week before that, giving *us* time to do a review day. Jiashuai may provide additional review days depending on how you feel.
- I'll also provide a few extra office hours before and after the days I'm gone.

Graph Review



- We're about to begin a few weeks on graphs algorithms, so let's begin with some review on definitions and graph traversals. I'll stay light on proofs today, because you likely saw this stuff in your discrete structures courses.
- A *graph* G = (V, E) is a set of *vertices* or *nodes* V and *edges* E. If G is undirected, each edge is a set of vertices, but I'll write uv. If G is directed, each edge is an ordered pair, but I'll write u ➡ v.
- This definition does not allow for loops or parallel edges, meaning we must work with *simple* graphs. Most of the algorithms we talk about extend to multigraph with almost no change.
- If uv is an edge, then u is a *neighbor* of v and vice versa.
- The degree of a vertex is the number of neighbors. If u ➡ v is a directed edge, then u is a

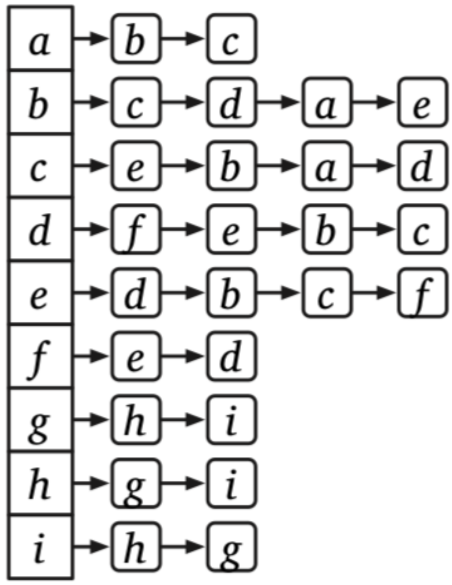
predecessor of v and v is a *successor* of u.

- The *in-degree* of a vertex is the number of predecessors and the *out-degree* is the number of successors.
- **[start skipping here]**
- A graph G' = (V', E') is a *subgraph* of G = (V, E) if V' subseteq V and E' subseteq E.
- A *walk* is a sequence of edges where each successive pair of edges share a vertex. A *path* is a walk that visits each vertex at most once.
- An undirected graph is *connected* if there is a walk between every pair of vertices. The *components* of a graph are its maximal connected subgraphs.
- A *cycle* is a walk that only repeats its first / last vertex and has at least one edge. A graph is *acyclic* or a *forest* if no subgraph is a cycle. A *tree* is a connected acyclic graph. A *spanning tree* of G is a subgraph of G that contains every vertex and is a tree. A *spanning forest* has one spanning tree per component of G.
- A directed graph is strongly connected if there is a directed walk between every ordered pair of vertices. A directed graph is acyclic if there is no directed cycle. I might say *DAG* to mean directed acyclic graph.
- **[end skipping]**
- When describing graph algorithms, we may use V or E to represent the *number* of vertices or edges in the input graph, i.e., this algorithm runs in time O(V + E). Yes, this is weird, and I personally prefer using n and m outside the classroom. For some reason, *both* textbooks do this V and E thing, and writing |E| is tedious, so we'll go with the flow.

Computer Representations

- So how do computers represent graphs?
- The two most common methods are the adjacency matrix and the adjacency list. The first lets you look up the existence of edges in constant time, but it uses Theta(V^2) space.

	a	b	c	d	e	f	g	h	i
a	0	1	1	0	0	0	0	0	0
b	1	0	1	1	1	0	0	0	0
c	1	1	0	1	1	0	0	0	0
d	0	1	1	0	1	1	0	0	0
e	0	1	1	1	0	1	0	0	0
f	0	0	0	1	1	0	0	0	0
g	0	0	0	0	0	0	0	1	0
h	0	0	0	0	0	0	1	0	1
i	0	0	0	0	0	0	1	1	0



- **[skip during lecture]** The *adjacency matrix* of G = (V, E) is a V x V matrix where A[i, j] = 1 if (i, j) \in E and 0 otherwise. If the graph is undirected, then A[i, j] = A[j, i] in every case. For simple graphs, meaning no loops, every diagonal entry A[i, i] = 0.
  - These use Theta(V^2) space, so it's only space-efficient for *dense* graphs.
  - But, we can decide in Theta(1) time if two vertices are adjacent.
  - Listing all neighbors of a vertex means searching its whole row or column in Theta(V) time.
- But generally, we'll use the *adjacency list* of G which is an array with one linked-list per vertex v, listing all of its neighbors. If G is directed, we store only successors of v. So in an undirected graph, each edge uv appears in the lists for both u and v. Directed edge u ➡ v appears exactly once and in u's list.
  - The space used is only Theta(V + E) so its space-efficient even for *sparse* graphs.
  - Listing the neighbors of vertex u takes O(1 + deg(u)) time since we only need to scan v's list.
  - But, we do need O(1 + deg(u)) time to decide if edge u ➡ v exists or O(1 + min{deg(u), deg(v)} time to find edge uv if the graph is undirected. In most graph algorithms, we never need to ask *if* an edge exists, only list edges coming out of certain vertices of interest.

BFS

- A common operation on graphs is to traverse or search its vertices and edges. In particular, we may ask if v is *reachable* from s, meaning there is a (directed) path from s to v.
- Probably the simplest traversal algorithm is the *breadth-first search* or BFS. It works by trying to visit vertices in order of their distance from s. To avoid repeating work, we *mark* vertices we've seen. Initially all vertices are unmarked.

- Here's one way to implement it based on Erickson. The CLRS implementation is somewhat different but still visits the same vertices.
- BFS(s):
  - put (emptyset, s) in a queue
  - while the queue is not empty
    - take (p, v) from the queue
    - if v is unmarked
      - mark v
      - parent(v) ← p
      - for each edge vw
        - put (v, w) into the queue
- Using induction, we can prove several facts about BFS:
  1. It marks every vertex reachable from s exactly once.
  2. The set of pairs (v, parent(v)) form a spanning tree on the *component* of G containing s, i.e., the set of vertices reachable from s.
  3. The paths in this spanning tree are shortest paths from s to their endpoints, where every edge has length 1.
- I want to focus a bit more on the analysis of this algorithm, though.
- That for loop is executed once per marked vertex, so at most V times.
- Therefore, each edge vw is put into the bag at most *twice*, once as (v, w) and once as (w, v). So we enqueue at most 2E times.
- And we can't take more out of the queue than we put in, so we dequeue at most 2E + 1 times.
- Queue operations take O(1) time each, so the total running time is O(V + E).
- If you're working with a directed graph, then you loop over edges *leaving* v and you get a spanning tree over vertices specifically reachable from s.
- So remember, if you need to compute shortest paths on an unweighted graph, USE BFS IT'S LINEAR TIME.
- Finally, Erickson remarks that this is just a special case of a generic graph search algorithm he calls WhateverFirstSearch. By replacing the queue with other data structures, you get other search algorithms, including Prim's algorithm for minimum spanning times , Dijkstra's algorithm for shortest paths, and a non-recursive version of depth-first search. We'll cover all of those algorithms in the next two weeks, starting with depth-first search today and some applications next Tuesday.

DFS

- Now, having said that about WhateverFirstSearch, probably the most common way to implement depth-first search or DFS it is to use recursion.

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

- We can extend this algorithm to mark all vertices in the graph by using a wrapper function.

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

- We still mark each vertex once and therefore handle each directed edge once, so the running time is O(V + E).

Preorder and Postorder

- The applications for DFS all come from the useful order in which it marks vertices.
- To see that, let's pass around a clock variable that increments every time we start or stop visiting a vertex.

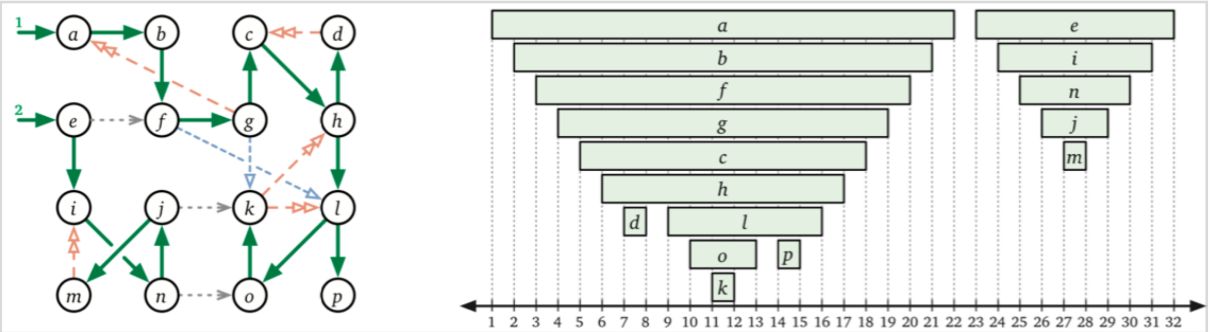
```
DFSALL(G):
  clock ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          clock ← DFS(v, clock)
```

```
DFS(v, clock):
  mark v
  clock ← clock + 1;  v.pre ← clock
  for each edge v→w
      if w is unmarked
          w.parent ← v
          clock ← DFS(w, clock)
  clock ← clock + 1;  v.post ← clock
  return clock
```

- We assign v.pre just after pushing v onto the recursion stack and assign v.post just before popping it from the stack.
  - v.pre is often called the *starting time* of v.
  - v.post is often called the *finishing time* of v.
  - and [v.pre, v.post] is called the *active interval* of v.
- So, because stack timelines are always disjoint or nested, [u.pre, u.post] and [v.pre, v.post] are either disjoint or nested. In fact, [u.pre, u.post] contains [v.pre, v.post] if and only if DFS(v) is called during the execution of DFS(u).
- And because we only make recursive calls when there are edges, there must be a directed path from u to v in this case. In particular, the set of vertices on the recursion stack form a

directed path in G.

- Here's and example of a depth-first search.



- Similar to rooted trees, we can use the v.pre labels to get a *preordering* of the vertices "abfgchdlokpeinjm" in that order, and the v.post labels to get a *postordering* "dkoplhcgfbamjnie" in that order.

Classifying Vertices and Edges

- So let's say we're in the middle of running a depth-first search. We can learn a lot about the structure of the graph by using this clock variable.
- Eventually, the algorithm will populate v.pre and v.post for every vertex v.
- But suppose we're midway through running DFS. Fix a vertex v and its eventual pre and post values. But consider the clock at the moment we pause the algorithm. v is
  - *new* if clock < v.pre (DFS(v) has not yet been called)
  - *active* if v.pre ≤ clock < v.post (DFS(v) has been called but not yet returned)
  - *finished* if v.post ≤ clock (DFS(v) has returned)
- Being active corresponds to a vertex being on the recursion stack. That means the active vertices form a directed path in G.
- In turn, using these definitions, we can partition the edges into four classes depending on how they interact with the depth-first search tree. Unlike vertices, these classes apply to a run of DFS, not a particular moment in time during the run. Consider edge u ➡ v and the moment when DFS(u) begins.
  - If v is new, then either we call DFS(v) directly when we iterate over u ➡ v, or another intermediate recursive call will mark v first. Either way, u.pre < v.pre < v.post < u.post.
    - If DFS(u) calls DFS(v) directly, u ➡ v is called a *tree edge*.
    - Otherwise, u ➡ v is called a *forward edge*.
  - If v is active, then v is on the stack, so v.pre < u.pre < u.post < v.post. G has a directed path from v to u.
    - u ➡ v is called a *back edge*.
  - If v is finished, then v.post < u.pre.
    - u ➡ v is called a *cross edge*.
  - Note that u.post < v.pre cannot happen, because we would add v to the stack before finishing with u.

- Again, this classification of edges depends upon the specific depth-first search tree we get, which depends upon the order in which we iterate over vertices and edges.