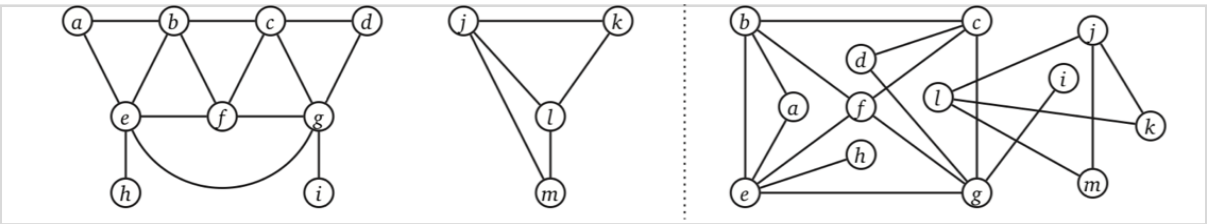
CS 4349.003.19F Lecture 12—October 7, 2019

Main topics for #lecture include #graph_representations and #graph_traversal.

Graphs

- Over the last few weeks, we saw different techniques for designing algorithms. I tried to focus on the techniques themselves, providing common examples as I went to show how the techniques have been used in the past and how you might use them to design your own algorithms in the future.
- Starting now and for the next several weeks, the focus will change somewhat. We'll be focusing almost exclusively on graph algorithms through the middle of November. There are fewer big "techniques" in this area, so I'll be shifting the emphasis to specific problems we may want to solve and the various algorithms used to solve those problems.
- I'll still ask you to do algorithm design in the homework assignments, but you'll focus more on using key properties of the algorithms from class and sometimes straight reductions to these algorithms that don't require coming up with anything too novel in the process.
- But before we get into any graph problems, we need to discuss some basic definitions and how I tend to use notation. Please look over Erickson Chapter 5 to get the full complement of definitions and notations. And please please please stop me to ask questions if I use some notation that is unfamiliar.
- So at its core, a graph is a collection of pairs—pairs of people, pairs of cities, pairs of web pages, pairs of anything. More formally, graph G = (V, E) consists of an *arbitrary* finite set (of anything) V called the *vertices*. The edges E are a set of pairs of elements of V.

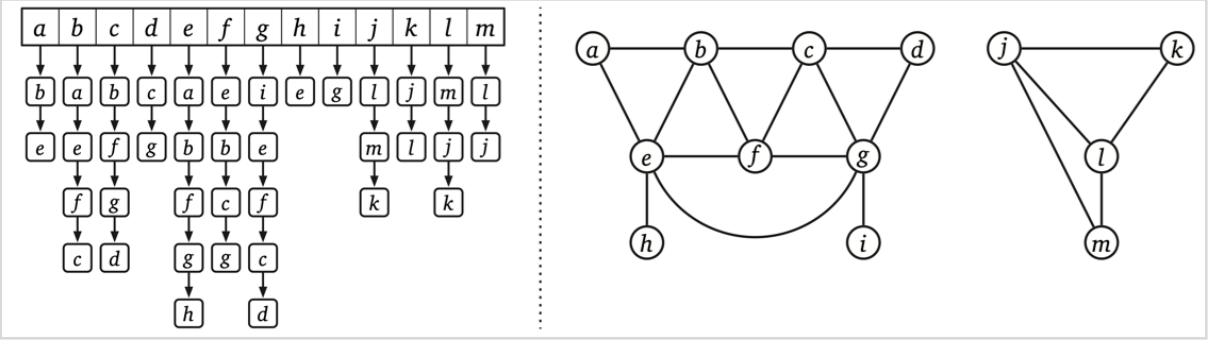


- For example, we may have these circles as the vertices and every edge is a pair of circles connected with a line segment. Again, the circles *are* the vertices in this case since the vertices can be a set of *anything*.
- If the graph is directed, I'll use u ➜ v for a directed edge. Otherwise, I like to use uv. If there is an containing u and v, we say u and v are *adjacent* and that uv is *incident* to u and v.
- 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 the *tail* of u ➜ v and v is the *head*. Also, 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.

- When describing graph algorithms, we may use V or E to represent the *number* of vertices or edges in the input graph, i.e., some algorithm may run 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.
- One key point I want to you to take away from this course is that graphs can represent tons of things, and they can be represented by tons of things.
- If you've seen shortest paths, then you're probably already aware that road networks can be represented by graphs. Each intersection is a vertex, and each segment of road between intersections is an edge.
- Genealogies can be represented by family "trees" where vertices are people and directed edges go to literal children. Iff you go far enough back, these start to look like directed acyclic graphs instead.
- More modern examples include graphs representing telecommunication networks and graphs representing social networks. *We* are the vertices in a social network like Facebook, and we share an undirected edge if we're friends. Twitter followers are best represented by a directed graph.
- Erickson provides many other examples of how graphs can represent or be represented by different phenomena in computing.

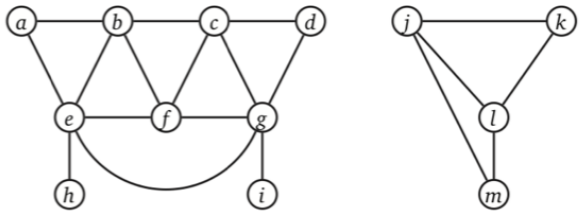
Data Structures

- We should discuss how graphs are actually stored on a computer so we can meaningfully talk about things like running time of algorithms.
- Most graphs are stored as either an *adjacency list* or an *adjacency matrix*. At a high level, both data structures are arrays indexed by the vertices, meaning we are essentially using the numbers 1 through V as the vertices.
- The most common data structure used is the *adjacency list*. It's an array of lists indexed by the vertices. If the graph is undirected, the list for u contains edge incident to (containing) u. If directed, the list contains every edge u ➜ v for which u is the tail.
- So undirected edges are stored twice, once per endpoint, and directed edges are stored once.
- Traditionally, the lists of edges are stored as singly linked lists.



- The big obvious reason to use adjacency lists is space: we can store it using Theta(V + E) space.
- Listing edges leaving a single vertex u is as fast as possible, we just traverse the list for u in O(deg(u)) time.
- But checking if an edge u ➜ v exists requires looking over u's entire list, still in O(deg(u)) time.
- Instead of using a linked list, we could use other data structures like hash tables. Now we get small space usage *and* fast lookup times.
- But in practice, hash tables are more complicated than we may like. It's hard to *guarantee* they're fast, and even then, you can only guarantee fast accesses *in expectation*.
- The other standard data structure is the adjacency matrix, a V x V matrix of 0s and 1s, usually stored as a 2D array A[1 .. V, 1 .. V]. If the graph is undirected, then A[u, v] is 1 if and only if uv is an edge. If directed, A[u, v] is 1 if and only if u ➜ v is an edge.

	a	b	c	d	e	f	g	h	i	j	k	l	m
a	0	1	0	0	1	0	0	0	0	0	0	0	0
b	1	0	1	0	1	1	0	0	0	0	0	0	0
c	0	1	0	1	0	1	1	0	0	0	0	0	0
d	0	0	1	0	0	0	1	0	0	0	0	0	0
e	1	1	0	0	0	1	1	1	0	0	0	0	0
f	0	1	1	0	1	0	1	0	0	0	0	0	0
g	0	0	1	1	1	1	0	0	1	0	0	0	0
h	0	0	0	0	1	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	1	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0	1	1	1
k	0	0	0	0	0	0	0	0	0	1	0	1	0
l	0	0	0	0	0	0	0	0	0	1	1	0	1
m	0	0	0	0	0	0	0	0	0	1	0	1	0



- Adjacency matrices are kind of bad at space. They require Theta(V^2) space no matter how many edges you have. Also, listing all the edges leaving a vertex u takes Theta(V) time, no matter how many edges u has.
- However, you can test if a particular edge uv exists in Theta(1) time.
- Here's a table summarizing these different data structures and the running times for common operations.

	Standard adjacency list (linked lists)	Fast adjacency list (hash tables)	Adjacency matrix
Space	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Test if $uv \in E$	$O(1+\min\{\deg(u),\deg(v)\}) = O(V)$	$O(1)$	$O(1)$
Test if $u \to v \in E$	$O(1+\deg(u)) = O(V)$	$O(1)$	$O(1)$
List v's (out-)neighbors	$\Theta(1+\deg(v)) = O(V)$	$\Theta(1+\deg(v)) = O(V)$	$\Theta(V)$
List all edges	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge uv	$O(1)$	$O(1)^*$	$O(1)$
Delete edge uv	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)^*$	$O(1)$

- Again, the hash table version of adjacency lists are rarely taught or used. Hash tables are complicated, easy to mess up, and if you want any guarantees on the running time, you have to use randomness. If you care about edge lookups and you know your graph is going to be dense anyway, it's actually faster in practice just to use the adjacency matrix.

Similarly, listing edges out of a vertex goes faster with an adjacency list.

- Also, for almost everything we'll do in this class, linked lists are actually good enough. We will very rarely if ever ask if a particular edge exists, but we will often iterate through all the edges incident to a single vertex.
- Finally, most uses of graphs in practice feel like working with either an adjacency list or adjacency matrix anyway. For example, if I'm reading a map, I'll usually try to figure out what roads leave an intersection by scanning around the intersection. Just like reading off the list of edges from a standard adjacency list.
- When you can get away with it, it's actually more efficient in many cases to run graph algorithms directly on the objects you care about instead of first building a separate graph. Like I said before, the thing being represented *is* the graph.
- Now, all that said, we will use standard adjacency lists exclusively unless I state otherwise.

Whatever-First Search

- So, let's finish today by actually discussing a non-trivial algorithm. First, a couple more definitions:
- 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.
- We'll look at a fundamental problem: given graph G and vertex s, what vertices are *reachable* from s. In other words, for what vertices v is there a path from s to v? In other other words, which vertices are in s's component? We'll stick with undirected graphs for today and most of Wednesday.
- You may have seen an algorithm for this problem called *depth-first search*. I'm going to introduce something a bit more abstract called *whatever-first search*.
- Whatever-first search stores a set of edges in some data structure I'll call a "bag". The only thing you need to know about the bag is that we can put stuff into it and pull stuff out. Specifically, we'll put edges we think may belong to a path from s to some other vertex into the bag. Since we don't need to reach s, we'll start with (emptyset, s) in the bag.
- And we'll mark vertices as we reach them so we don't keep searching again from those same vertices.

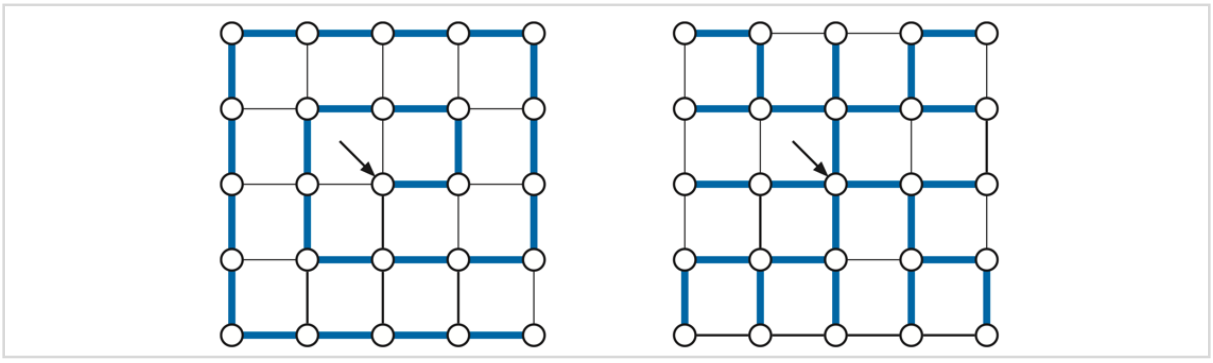
```
WHATEVERFIRSTSEARCH(s):
  put (∅, s) in bag
  while the bag is not empty
      take (p, v) from the bag                (⋆)
      if v is unmarked
          mark v
          parent(v) ← p
          for each edge vw                    (†)
              put (v, w) into the bag         (⋆⋆)
```

- Just a couple definitions so we can discuss what this algorithm does: A *cycle* is a walk that only repeats its first / last vertex and has at least one edge. A *tree* is a connected graph with no cycle. A *spanning tree* of graph G is a subgraph of G that contains every vertex and is a tree.
- Lemma: WhateverFirstSearch(s) marks every vertex reachable from s and only those vertices. Moreover, the set of pairs (v, parent(v)) with parent(v) ≠ emptyset defines a spanning tree of the component containing s.
- Proof:
  - First, everything is marked at most once thanks to that if statement.
  - Now, I'll prove we mark every reachable vertex v by induction on the shortest path distance from s to v. Doing induction over path length is pretty common when proving correctness of graph algorithms.
  - WhateverFirstSearch certainly marks s since we put it in the bag right away and immediately take it out. If v is reachable, there is a shortest path s ➔ … ➔ u ➔ v. Inductively, vertex u is marked, and then (u, v) is put in the bag. But later (u, v) is pulled out and v is marked if it isn't marked already.
  - Now to prove every marked vertex is reachable. Every pair (v, parent(v)) is an edge in the graph. I claim v ➔ parent(v) ➔ parent(parent(v)) ➔ … eventually leads to s by induction on the order in which vertices were marked. s leads back to s. For any other vertex v, parent(v) was marked before v. Because parent(v) ➔ parent(parent(v))… leads back to s, v ➔ parent(v) ➔ parent(parent(v))… leads back to s.
  - So, we mark exactly the reachable vertices. Further, the parent edges form a connected subgraph containing s and all other marked vertices. Every marked vertex except s has a unique parent, so the number of pairs is one less than the number of marked edges. A connected subgraph with one fewer edge than there are vertices is a tree (you can prove this statement using induction also!)

Analysis

- The time it takes to do the search depends upon what kind of bag we use. Say it takes T time to insert or remove something from the bag.
- The t for loop is executed once per marked vertex, so at most V times total.

- Each edge uv is put in the bag twice, once as (u, v) and once as (v, u) so the * * line runs at most 2E times.
- We can't take more out of the bag than we put in, so * runs at most 2E + 1 times.
- Assuming the graph is stored with an adjacency list, we can run that for loop for v in constant time per edge vw. The running time is O(V + ET).
- All that said, the bag we choose has some important consequences and can lead to algorithms you may have seen before.
- Suppose we implement the bag as a stack so we pull things out in the opposite order we add them. This leads to a depth-first search and *depth-first spanning tree*. The shape depends on the starting vertex and what order we add edges to the stack, but they tend to be long and skinny. We'll study their properties more next week. Pushing and popping from a stack takes O(1) time per operation, so the total running time is O(V + E).



- If we use a queue instead we get a breadth-first search and *breadth-first spanning tree*. Again, the exact look depends on the starting point and order we push edges, but these are short and bushy. In a couple weeks, we'll see that they contain shortest paths. Pushing and pulling takes O(1) time each, so the total running time is O(V + E).
- We could also use a priority queue for the bag to get what Erickson calls a *best-first search*. If we use a min-heap for our operations, then each insertion and removal takes O(log E) time for O(V + E log E) time total. The spanning tree we get depends upon what keys we use for the priority queue. One choice leads to minimum spanning trees. Another leads to shortest paths trees. A third choice leads to a so-called *widest path tree*. Again, we'll cover at least the first two of these trees over the next couple weeks.

- On Wednesday, we'll discuss some applications of whatever-first searching and I'll give you an opportunity to provide me with some mid-semester feedback.