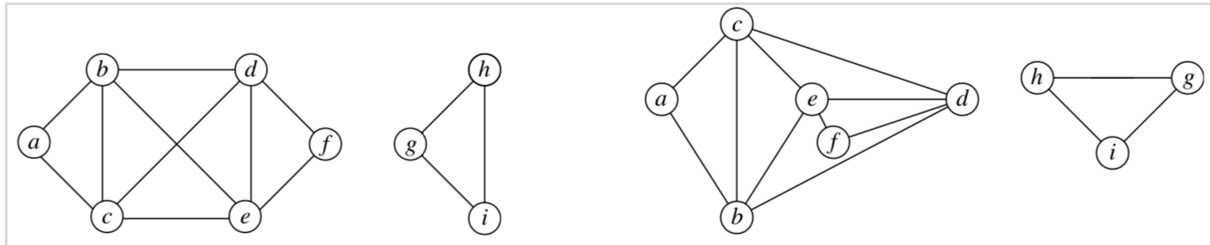
CS 4349 Lecture—October 9th, 2017

Main topics for #lecture include #graph_basics, and #graph_traversal.

Prelude

- Homework 5 due Wednesday, October 11th.

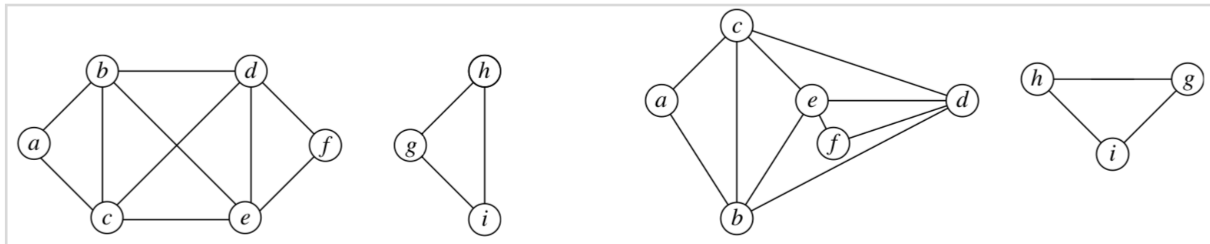
Graph Review



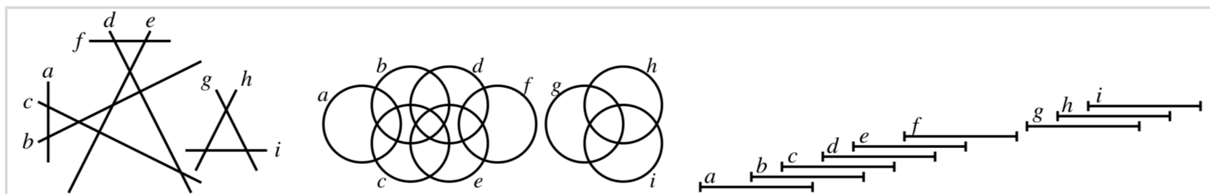
- A *graph* G = (V, E) is a set of *vertices* or *nodes* V and *edges* E. If G is undirected, each edge is a set of vertices, but I'll write uv. If G is directed, each edge is an ordered pair, but I'll write u ➔ v.
- This definition does not allow for loops or parallel edges, meaning we must work with *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.
- 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.
- 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).

How Humans Use and Represent Graphs

- Usually, we represent graphs using an *embedding* where each vertex is mapped to a point in the plane (drawn as a small circle) and edge edge is mapped to a curve or line segment between its vertex's points.
- The same graph can have multiple embeddings. The embedding and the graph itself are not the same thing. The embedding is merely one of many representations of the graph.



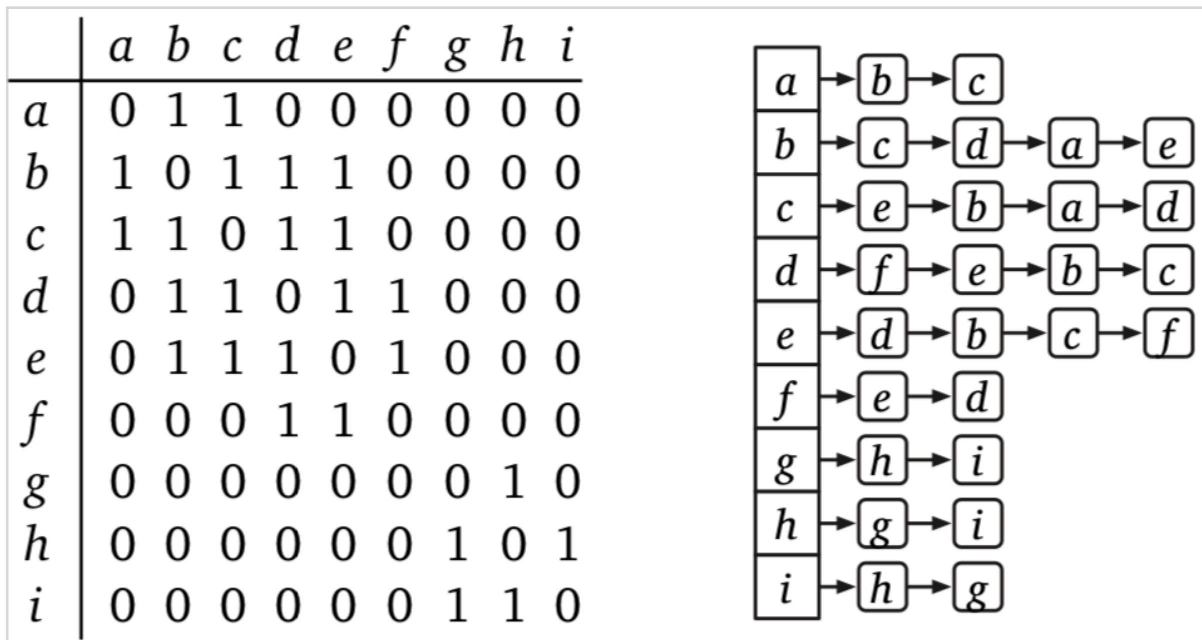
- But embeddings are not the only way to represent graphs, and these drawings are not the only things graphs represent.
- We can use graphs to represent pairwise relationships between any kind of objects. And that means we can look at the objects themselves to get a representation of the graph.
- Examples:
    - The *intersection graph* between geometric objects maps objects to vertices and has an edge for any pair of intersecting objects. The same graph represented using those embeddings is the intersection graph for these three sets



        - In particular, we call it the *interval graph* if the graph is the intersection graph of line segments on the real line.
    - The *dependency graph* of a recursive algorithm is a directed graph where the subproblems are vertices and there is a directed edge from one subproblem to another if you must evaluate the second to solve the first. These graphs are acyclic if your algorithm is guaranteed to terminate.
    - The *configuration graph* describes games, puzzles, or mechanisms like tic-tac-toe, checkers, or the Rubik's Cube. You have one vertex for each configuration and an edge between two configurations if you can move from one to the other in a single move.

Computer Representations

- But how to computers represent graphs?



	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

- 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.
- The *adjacency list* of G 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 edges appear once.
  - The space used is only Theta(V + E) so its space-efficient even for *sparse* graphs.
  - Listing the neighbors of vertex v takes O(1 + deg(v)) time since we only need to scan v's list.
  - But, we 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.
- There are fancier data structures we can use, but those are the most common ones. Adjacency matrices make edge lookups really simple and fast. But in most algorithms, we never actually ask *if* an edge exists, so we may as well use adjacency lists in almost every case.
- One neat consequence of working with the basic data structures is that we can apply many graph algorithms to certain representations by pretending we are working with one of the data structures directly.
  - For intersection graphs, we can pretend to work with an adjacency matrix just by checking for intersections every time we'd normally look at an entry in the matrix.
  - For configuration graphs, we can pretend to work with an adjacency list assuming we can figuring out all the moves out of a particular configuration in constant time per

configuration.

Graph Traversals

- But enough talking about graph data structures, let's use them to do something.
- Given: undirected graph G.
- We want to visit every vertex in G.
- You may have seen something called *depth-first search*. It can be written iteratively or recursively.
- Both versions are passed a *source* vertex s. The iterative algorithm using a stack that is initially empty.

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

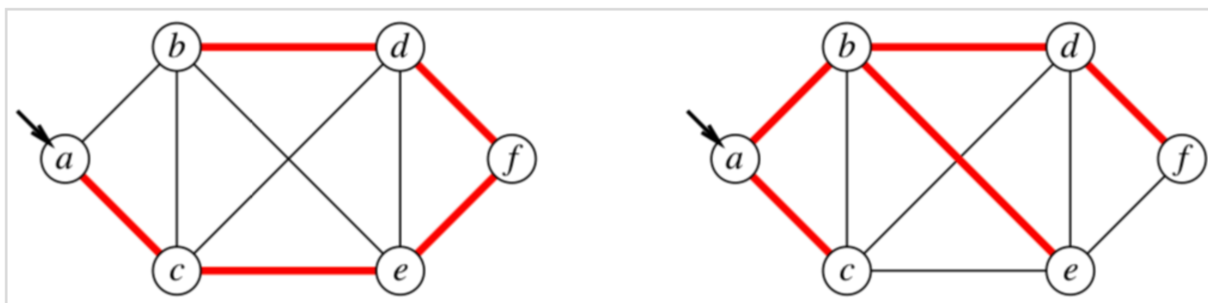
```
IterativeDFS(s):
    Push(s)
    while the stack is not empty
        v ← Pop
        if v is unmarked
            mark v
            for each edge vw
                Push(w)
```

- But depth-first search is just one of many similar graph traversal algorithms. These algorithms use a "bag" that takes stuff in and lets you pull stuff out. The only difference between the algorithms is the implementation of the bag.
- Here's a version of that 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.
- Proof:
  - The algorithm marks s.
  - Let (s, …, u, v) be the path from s to v with the minimum number of edges.
  - The algorithm marks u by induction on the shortest path distances from s. It adds (u, v)

to the bag, and later pulls (u, v) out, marking v if it hasn't been marked already. So every vertex gets marked.
  - And every vertex is marked at most once thanks to that if statement.
  - For every vertex v, (v, parent(v), parent(parent(v)), …) leads back to s, so parent edges form a connected graph. The graph has V - 1 edges so it's a tree.
- 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 exact 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 on Wednesday.



- 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).
  - 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.