CSE 332: Data Structures and Parallelism

Fall 2022

Anjali Agarwal

Lecture 18: Graph Theory

Announcements

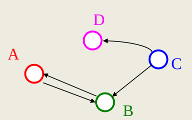
- Upcoming lectures
  - ~~Intro to graphs~~
  - Topological Sort
  - Parallelism (3 lectures)
  - Concurrency (2 lectures)

Graphs

A formalism for representing binary relationships between objects

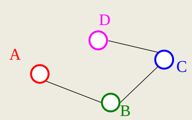
- Graph **G = (V,E)**
  - *Set of vertices*: $V = \{v_1, v_2, \ldots, v_n\}$
  - *Set of edges*: $E = \{e_1, e_2, \ldots, e_m\}$

Directed



V = {A, B, C, D}
E = {(C, B),(A, B),(B, A),(C, D)}

Undirected



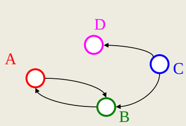
V = {A, B, C, D}
E = {{C, B},{A, B},{C, D}}

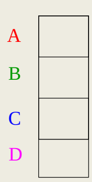
What's the data structure?

Common query: which edges are adjacent to a vertex

Representation 2: Adjacency List

A list (array) of length **|V|** in which each entry stores a list (linked list) of all adjacent vertices





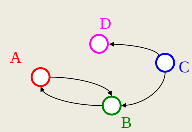
Runtimes:
Iterate over vertices?
Iterate over edges?
Iterate edges adj. to vertex?
Existence of edge?

Space requirements?

Best for what kinds of graphs?

Representation 1: Adjacency Matrix

A **|V| x |V|** matrix **M** in which an element **M[u,v]** is true if and only if there is an edge from **u** to **v**



	A	B	C	D
A				
B				
C				
D				

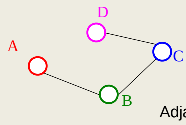
Runtimes:
Iterate over vertices?
Iterate over edges?
Iterate edges adj. to vertex?
Existence of edge?

Space requirements?

Best for what kinds of graphs?

Representing Undirected Graphs

What do these reps look like for an undirected graph?



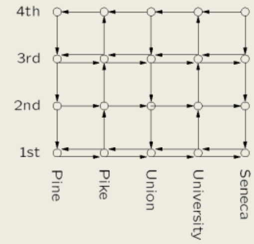
Adjacency matrix:

	A	B	C	D
A				
B				
C				
D				

Adjacency list:

A	
B	
C	
D	

Some Applications: Bus Routes in Downtown Seattle

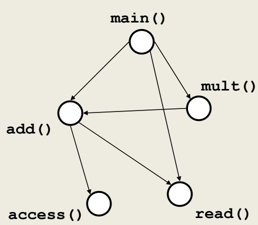


If we're at 3rd and Pine, how can we get to 1st and University using Metro?

How about 4th and Seneca?

Directed Acyclic Graphs (DAGs)

- DAGs are directed graphs with no (directed) cycles.

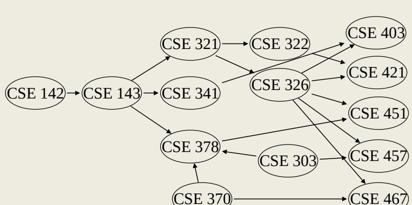


|E| and |V|

- How many edges |E| in a graph with |V| vertices?
- What if the graph is directed?
- What if it is undirected and connected?
- Some (semi-standard) terminology:
  - A graph is *sparse* if it has O(|V|) edges (upper bound).
  - A graph is *dense* if it has $\Theta(|V|^2)$ edges.

Application: Topological Sort

- Given a graph, `G = (V,E)`, output all the vertices in `V` sorted so that no vertex is output before any other vertex with an edge to it.



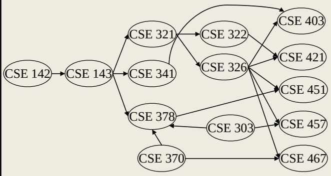
What kind of input graph is allowed?

Is the output unique?

Topological Sort: Take One

1. Label each vertex with its *in-degree* (# inbound edges)
2. **While** there are vertices remaining:
   a. Choose a vertex *v* of *in-degree zero*; output *v*
   b. Reduce the in-degree of all vertices adjacent to *v*
   c. Remove *v* from the list of vertices

Runtime:



142
143
321
341
378
370
322
326

303
403
421
451
457
467


```
void topsort(){
  Vertex v, w;

  labelEachVertexWithItsInDegree();

  for (int counter=0; counter < NUM_VERTICES; counter++){
          v = findNewVertexOfDegreeZero();

          v.topologicalNum = counter;
          for each w adjacent to v
                w.indegree--;
  }
}
```


Topological Sort: Take Two

1. Label each vertex with its in-degree
2. Initialize a queue *Q* to contain all in-degree zero vertices
3. While *Q* not empty
   a. *v* = *Q*.dequeue; output *v*
   b. Reduce the in-degree of all vertices adjacent to *v*
   c. If new in-degree of any such vertex *u* is zero *Q*.enqueue(*u*)

Runtime:

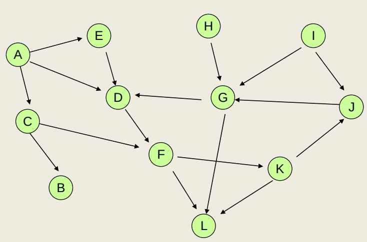

```
topsort(){
  Queue q(NUM_VERTICES);
  Vertex v, w;

  labelEachVertexWithItsIn-degree();

  q.makeEmpty();
  for each vertex v
    if (v.indegree == 0)
      q.enqueue(v);

  while (!q.isEmpty()){
    v = q.dequeue();
    v.topologicalNum = ++counter;
    for each w adjacent to v
      if (--w.indegree == 0)
        q.enqueue(w);
  }
}
```

initialize the queue

get a vertex with indegree 0

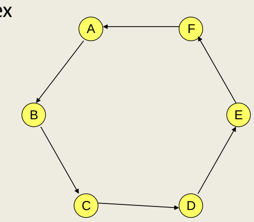
insert new eligible vertices

Find a topological order for the following graph



If a graph has a cycle, there is no topological sort

- Consider the first vertex on the cycle in the topological sort
- It must have an incoming edge



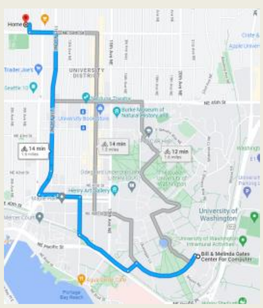
Lemma: If a graph is acyclic, it has a vertex with in degree 0

- Proof:
  - Pick a vertex v_1, if it has in-degree 0 then done
  - If not, let (v_2, v_1) be an edge, if v_2 has in-degree 0 then done
  - If not, let (v_3, v_2) be an edge . . .
  - If this process continues for more than n steps, we have a repeated vertex, so we have a cycle

Shortest Paths Problem

- Given a directed graph with edge costs and a starting vertex s, find the minimum cost path from s to every other vertex in the graph.
- Future results
  - Dijkstra's algorithm solves the shortest paths problems if all costs are non-negative
  - Bellman-Ford's algorithm solves the shortest paths problem if costs are allowed to be negative
    - Project 3 implements, and parallelizes Bellman-Ford

Find the shortest path



The Shortest Path Problem

Given a graph *G*, and vertices *s* and *t* in *G*, find the shortest path from *s* to *t*.

Two cases: weighted and unweighted.

For a path $p = v_0\ v_1\ v_2 \ldots v_k$

- *unweighted length* of path $p = k$ (a.k.a. *length*)
- *weighted length* of path $p = \sum_{i=0..k-1} c_{i,i+1}$ (a.k.a. *cost*)

We will assume the graph is directed

Single Source Shortest Paths (SSSP)

Given a graph *G* and vertex *s*, find the shortest paths from *s* to all vertices in G.

- How much harder is this than finding single shortest path from s to t?
  - Most algorithms will have to find the shortest path to every vertex in the graph in the worst case
    - Although may stop early in some cases