CSE 373: Data Structures and Algorithms

Lecture 15: Graph Data Structures, Topological Sort, and Traversals (DFS, BFS)

Instructor: Lilian de Greef
Quarter: Summer 2017

Today:

- Announcements
- Graph data structures
- Topological Sort
- Graph Traversals
  - Depth First Search (DFS)
  - Breadth First Search (BFS)

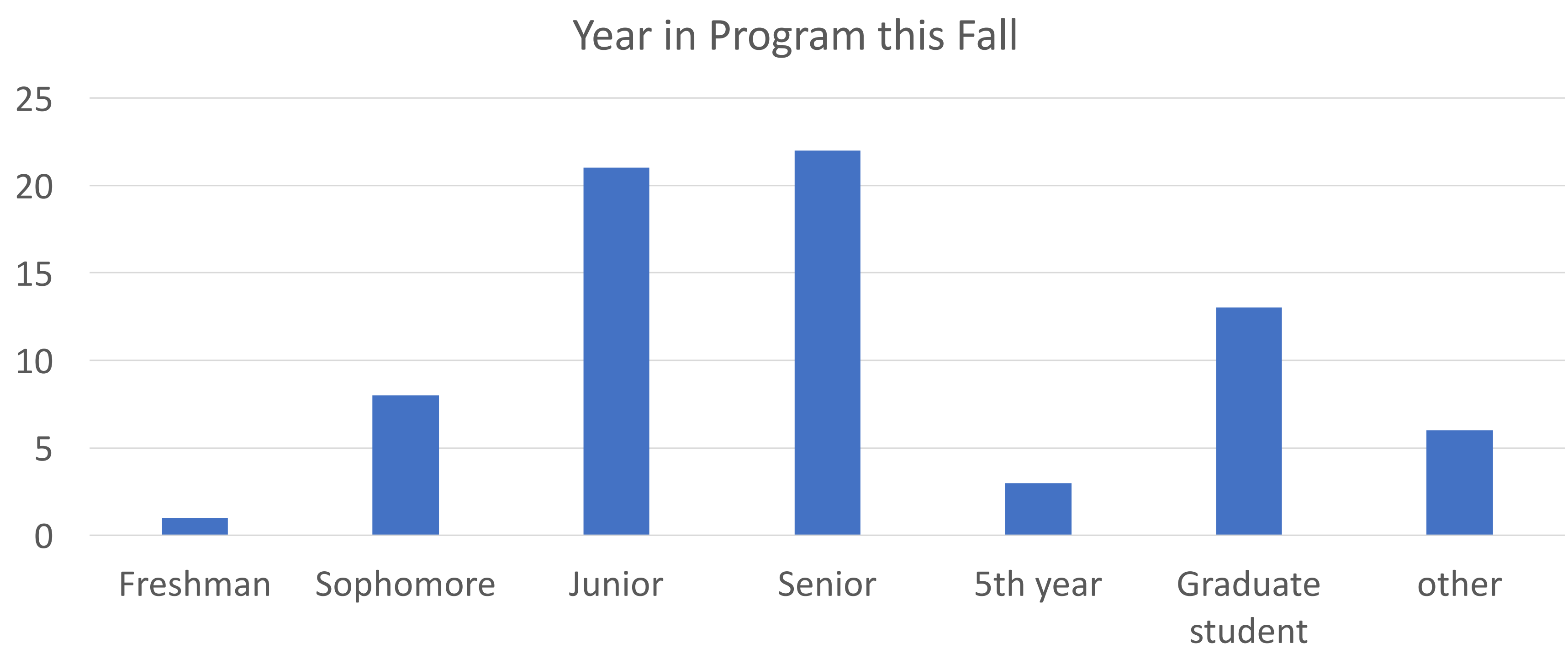
Announcement: Received Course Feedback

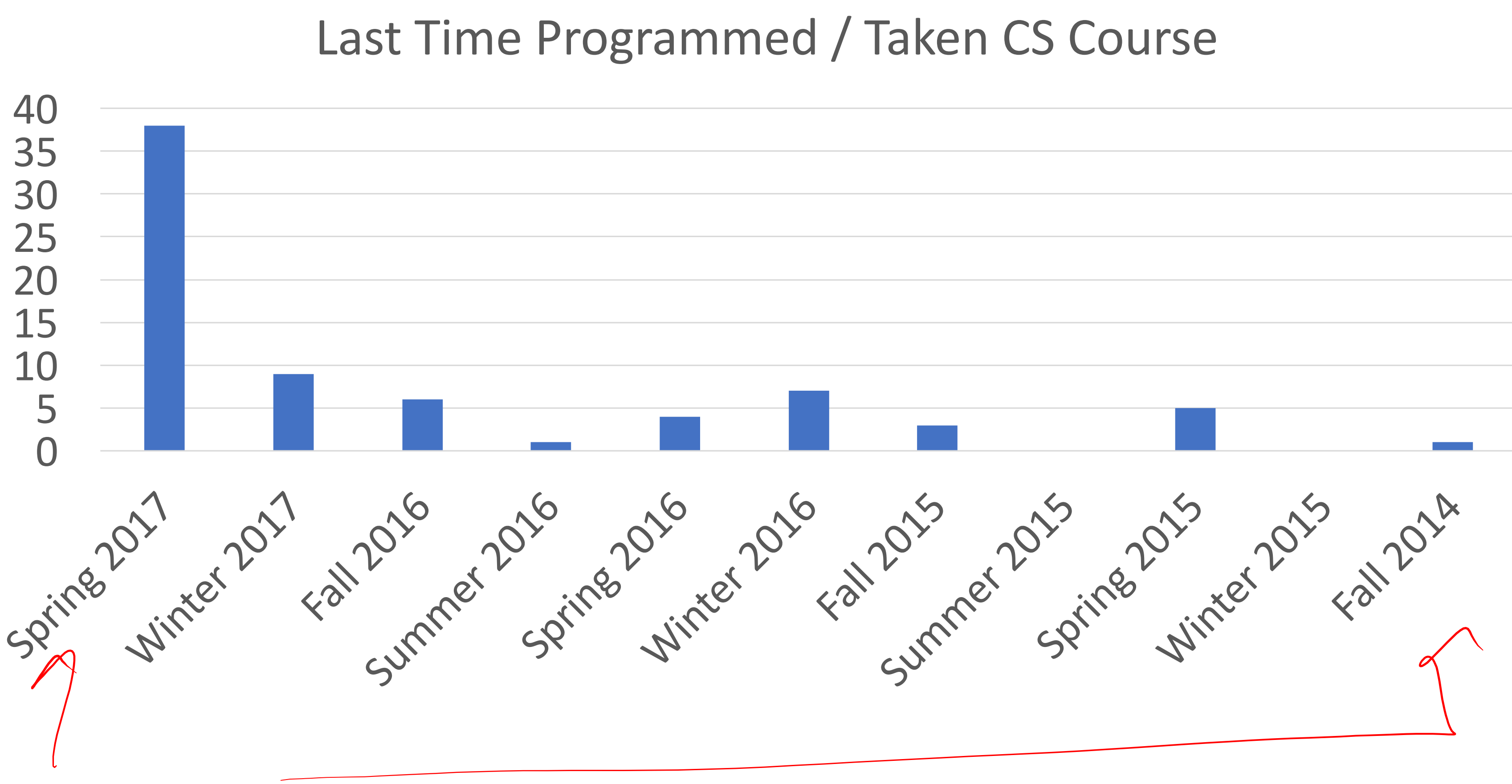
What’s working well:

- Walking through in-class examples
- Posted, printed, and annotated slides
- Interactive questions & in-class partner discussion

Things to address:

- Amount to write on printed slides
- Why using polling system for in-class exercises
- Concern about not getting through entire slide deck


Year in Program this Fall
25
20
15
10
5
0
Freshman
Sophomore
Junior
Senior
5th year
Graduate student
other



Last Time Programmed / Taken CS Course
40
35
30
25
20
15
10
5
0
Spring 2017
Winter 2017
Fall 2016
Summer 2016
Spring 2016
Winter 2016
Fall 2015
Summer 2015
Spring 2015
Winter 2015
Fall 2014

Represented Majors

- Engineering
- Math
- Science
- Informatics
- Geology
- Spanish
- Asian Language
- Pre-major
- And more!

Graph Data Structures

A couple of different ways to store adjacencies

What is the Data Structure?

- So graphs are really useful for lots of data and questions
  - For example, "what's the lowest-cost path from x to y"

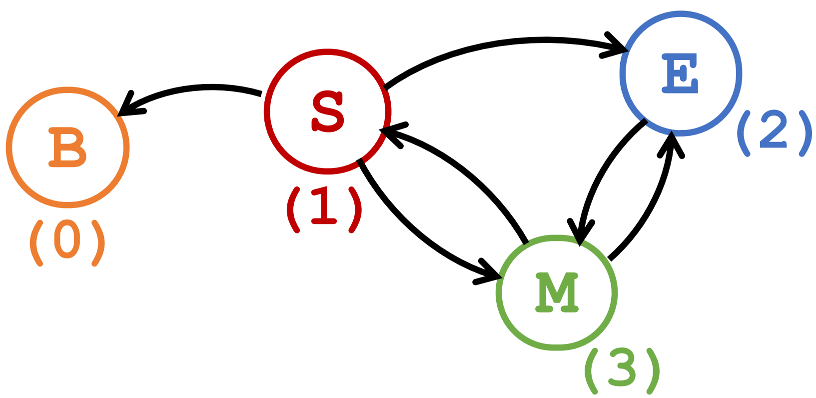
- But we need a data structure that represents graphs

- The "best one" can depend on:
  - Properties of the graph (e.g., dense versus sparse)
  - The common queries (e.g., "is `(u,v)` an edge?" versus "what are the neighbors of node **`u`**?")

- So we'll discuss the two standard graph representations
  - **Adjacency Matrix** and **Adjacency List**
  - Different trade-offs, particularly time versus space

Adjacency Matrix

- Assign each node a number from `0` to `|V|-1`
- A `|V|` x `|V|` matrix (i.e., 2-D array) of Booleans (or 1 vs. 0)
  - If `M` is the matrix, then `M[u][v] == true` means there is an edge from `u` to `v`



	0	1	2	3
0	F	F	F	F
1	T	F	T	T
2	F	F	F	T
3	F	T	T	F

Adjacency Matrix Properties

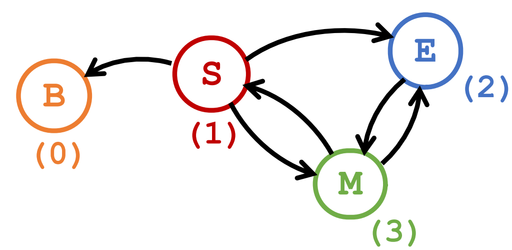
- Running time to:
  - Get a vertex's out-edges: $O(|V|)$
  - Get a vertex's in-edges: $O(|V|)$
  - Decide if some edge exists: $O(1)$
  - Insert an edge: $O(1)$
  - Delete an edge: $O(1)$
- Space requirements:

  $O(|V|^2)$
- Best for ~~sparse~~ or dense graphs?

$|V|$

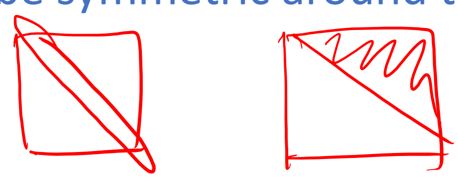
	0	1	2	3
0	F	F	F	F
1	T	F	T	T
2	F	F	~~T~~ F	T
3	F	T	T	F

$|V|$



Adjacency Matrix Properties

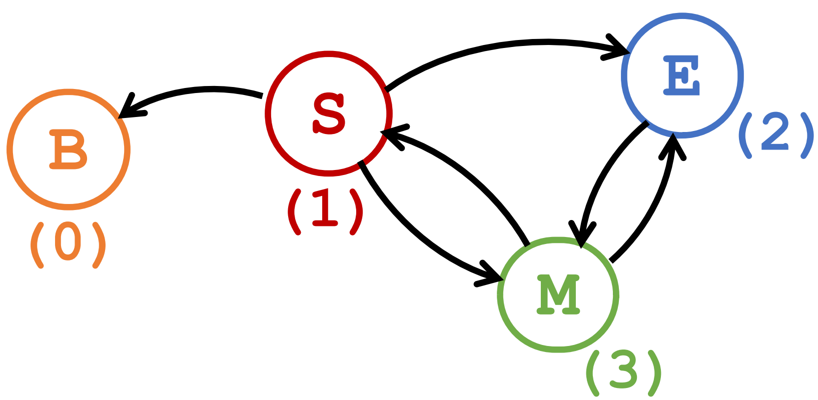
- How will the adjacency matrix vary for an *undirected graph*?
  - Undirected will be symmetric around the diagonal

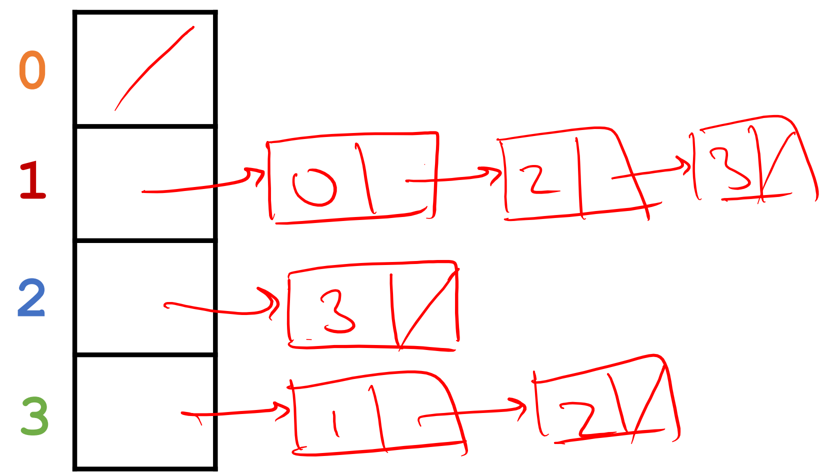


- How can we adapt the representation for *weighted graphs*?
  - Instead of a Boolean, store a number in each cell
  - Need some value to represent 'not an edge'
    - In *some* situations, 0 or -1 works

Adjacency List

- Assign each node a number from `0` to `|V|-1`
- An array of length |V| in which each entry stores a list of all adjacent vertices (e.g., linked list)

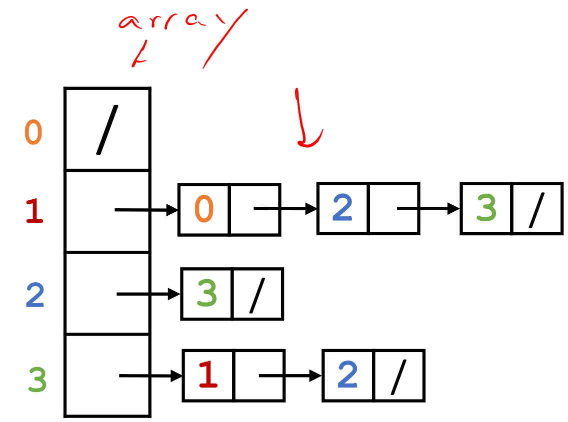




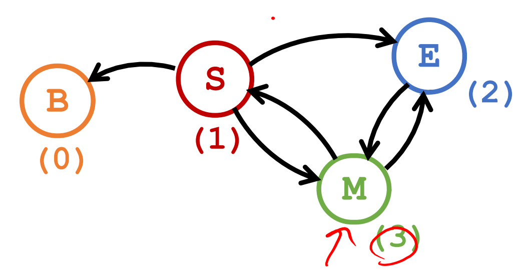
Adjacency List Properties

$O(|V|^2)$

worst case: $|E| == |V|^2$



- Running time to:
  - Get all of a vertex's out-edges:
    $O(d)$ where *d* is out-degree of vertex
  - Get all of a vertex's in-edges:
    $O(|V| + |E|)$ (but could keep a second adjacency list for this!)
  - Decide if some edge exists:
    $O(d)$ where *d* is out-degree of source
  - Insert an edge:
    $O(1)$ (unless you need to check if it's there)
  - Delete an edge:
    $O(d)$ where *d* is out-degree of source
- Space requirements:
  $O(|V| + |E|)$



Best for sparse or ~~dense~~ graphs?

$O(|V| + |E|) < O(|V|^2)$

Algorithms

Okay, we can represent graphs

Now we'll implement some useful and non-trivial algorithms!

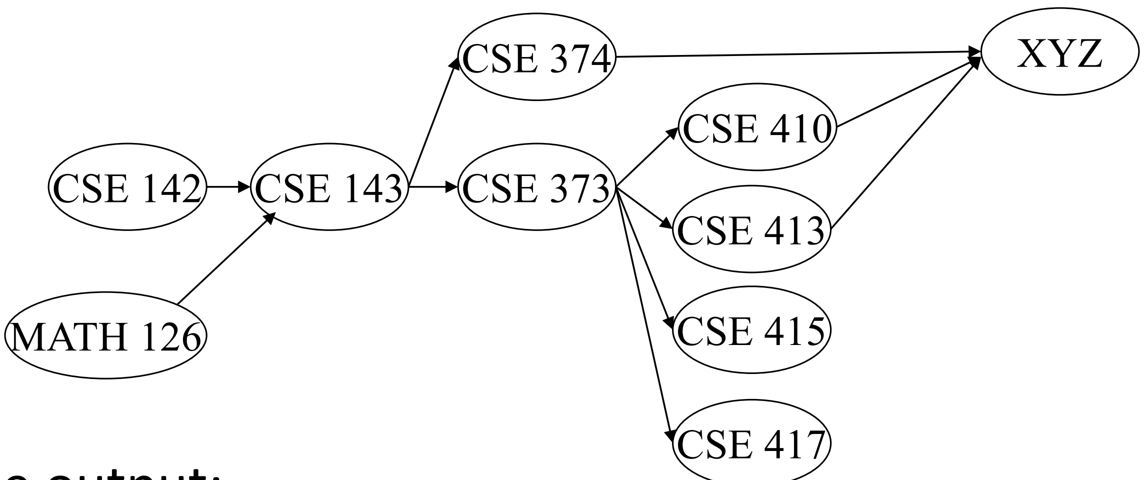
- Topological Sort
- Shortest Paths
  - Related: Determining if such a path exists
  - Depth First Search
  - Breadth First Search

Graphs: Topological Sort

Ordering vertices in a DAG

Topological Sort

Topological sort: Given a DAG, order all the vertices so that every vertex comes before all of its neighbors



One example output:

126, 142, 143, 374, 373, 417, 410, 413, XYZ, 415

Questions and comments

directed acyclic graphs

- Why do we perform topological sorts only on DAGs?

  cycle → no correct answer

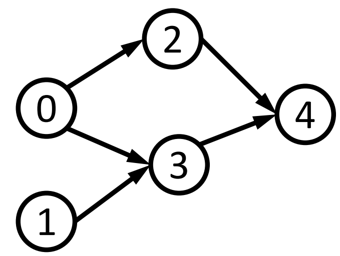
- Is there always a unique answer?

  Depends on graph

- Do some DAGs have exactly 1 answer?

  Yes, e.g. all lists

- Terminology: A DAG represents a **partial order** and a topological sort produces a **total order** that is consistent with it



A few of its uses

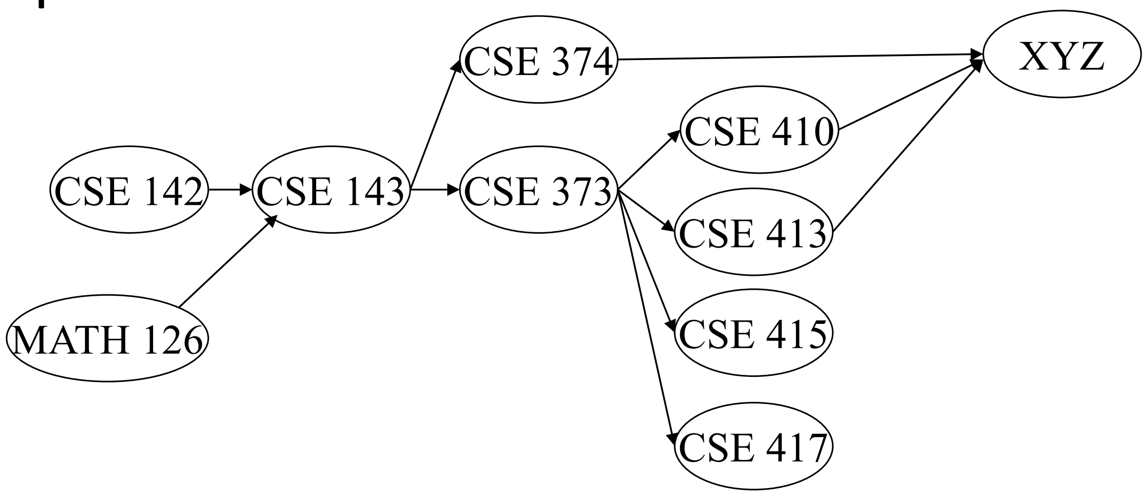
- Figuring out how to graduate
- Computing an order in which to recompute cells in a spreadsheet
- Determining an order to compile files using a Makefile
- In general, taking a dependency graph and finding an order of execution

A First Algorithm for Topological Sort

1. Label ("mark") each vertex with its in-degree
   - Could "write in a field in the vertex"
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex **v** with in-degree of 0
   b) Output **v** and *conceptually* remove it from the graph
   c) For each vertex **u** adjacent to **v** (i.e. **u** such that (**v**,**u**) in **`E`**), decrement the in-degree of **u**

Example

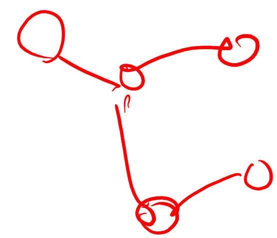


Node:	126	142	143	374	373	410	413	415	417	XYZ
Removed?	X	X	X	X	X	X	X	X	X	X
In-degree:	0	0	~~2~~ ~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~1~~ 0	~~3~~ ~~2~~ ~~1~~ 0

Output:

126
142
143
374
373
417
410
413
XYZ
415

Notice



- Needed a vertex with in-degree 0 to start
  - Will always have at least 1 because no cycles!
- Ties among vertices with in-degrees of 0 can be broken arbitrarily
  - Can be more than one correct answer, by definition, depending on the graph

Running time?

```
labelEachVertexWithItsInDegree();
for(i = 0; i < numVertices; i++){
  v = findNewVertexOfDegreeZero();
  put v next in output
  for each u adjacent to v
    u.indegree--;
}
```

worst case:
$|E| = |V|^2$
(dense)

- What is the worst-case running time?
  - Initialization $O(|V|+|E|)$ (assuming adjacency list)
  - Sum of all find-new-vertex $O(|V|^2)$ (because each $O(|V|)$)
  - Sum of all decrements $O(|E|)$ (assuming adjacency list)
  - So total is $O(|V|^2)$ – not good for a sparse graph!

Doing better

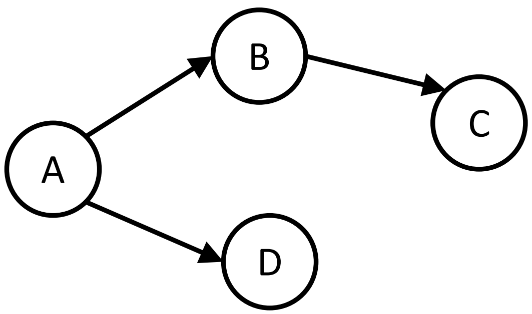
The trick is to avoid searching for a zero-degree node every time!

- Keep the "pending" zero-degree nodes in a list, stack, queue, bag, table, or something
- Order we process them affects output but not correctness or efficiency, provided that add/remove are both $O(1)$

Using a queue:

1. Label each vertex with its in-degree, enqueue 0-degree nodes
2. While queue is not empty
   a) **v** = dequeue()
   b) Output **v** and remove it from the graph
   c) For each vertex **u** adjacent to **v** (i.e. **u** such that (**v**,**u**) in **E**), decrement the in-degree of **u**, if new degree is 0, enqueue it

Example: Topological Sort Using Queues



Node	A	B	C	D
Removed?	X	X	X	X
In-degree	0	~~1~~ 0	~~1~~ 0	~~2~~ 0

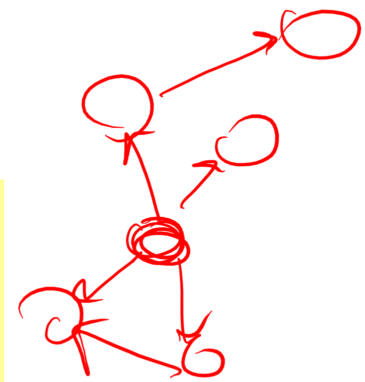
Queue: ~~A~~ ~~B~~ ~~D~~ ~~C~~

Output: A B D C

The trick is to avoid searching for a zero-degree node every time!

1. Label each vertex with its in-degree, enqueue 0-degree nodes
2. While queue is not empty
   a) **v** = dequeue()
   b) Output **v** and remove it from the graph
   c) For each vertex **u** adjacent to **v** (i.e. **u** such that (**v**,**u**) in **E**), decrement the in-degree of **u**, if new degree is 0, enqueue it

Running time?



```
labelAllAndEnqueueZeros();
 for(i=0; ctr < numVertices; ctr++){
    v = dequeue();
    put v next in output
   for each u adjacent to v {
     u.indegree--;
     if(u.indegree==0)
       enqueue(u);
   }
}
```

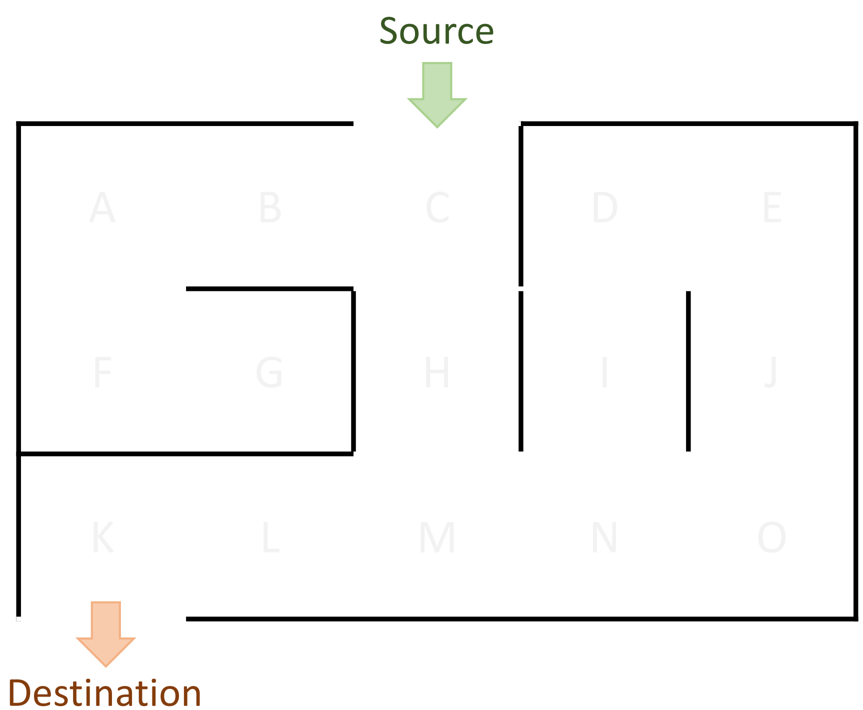
- What is the worst-case running time?
  - Initialization: $O(|V|+|E|)$ (assuming adjacency list)
  - Sum of all enqueues and dequeues: $O(|V|)$
  - Sum of all decrements: $O(|E|)$ (assuming adjacency list)
  - Total: $O(|V|+|E|)$ – much better for sparse graph!

Graph Traversals

Depth- and Breadth- First Searches!

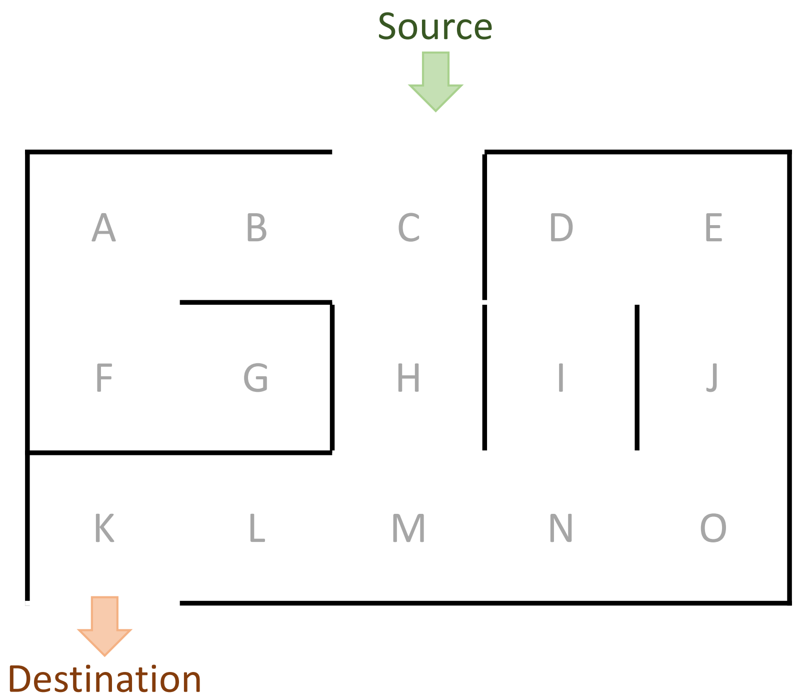
Introductory Example: Graph Traversals

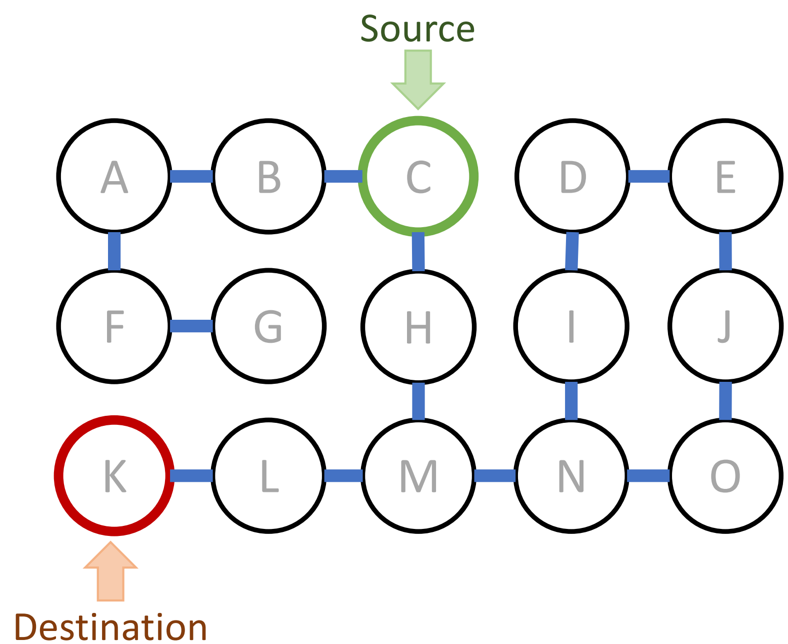
How would a computer systematically find a path through the maze?



Note: under the hood, we're using a graph to represent the maze

In graph terminology: find a path (if any) from one vertex to another.





Find a path (if any) from one vertex to another.

Let's try keeping track **recursively**

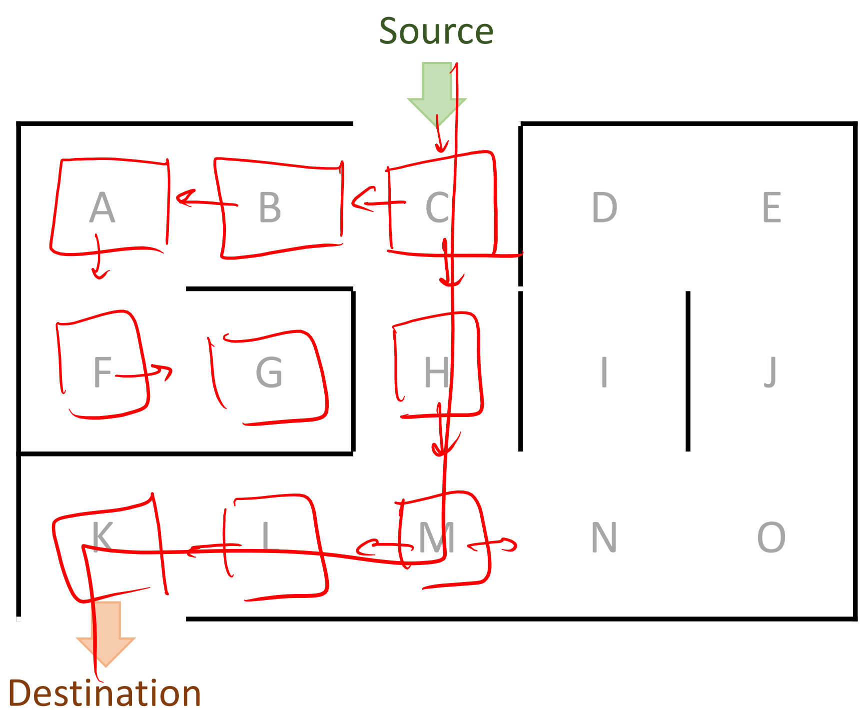
Idea: Repeatedly explore and keep track of adjacent vertices.

Mark each vertex we visit, so we don't process each more than once.

cycles!

Store as additional variable in vertices

DFS!



C
~~B~~
~~A~~
~~F~~
~~G~~

C
H
M
L
K

Depth First Search (DFS)

Depth First Search (**DFS**):

Explore as far as possible along each branch before backtracking

Repeatedly explore adjacent vertices using recursion or a stack

Mark each vertex we visit, so we don't process each more than once.

Example pseudocode:

```
DFS(Node start) {
  mark and process start
  for each node u adjacent to start
    if u is not marked
      DFS(u)
}
```

Find a path (if any) from one vertex to another.

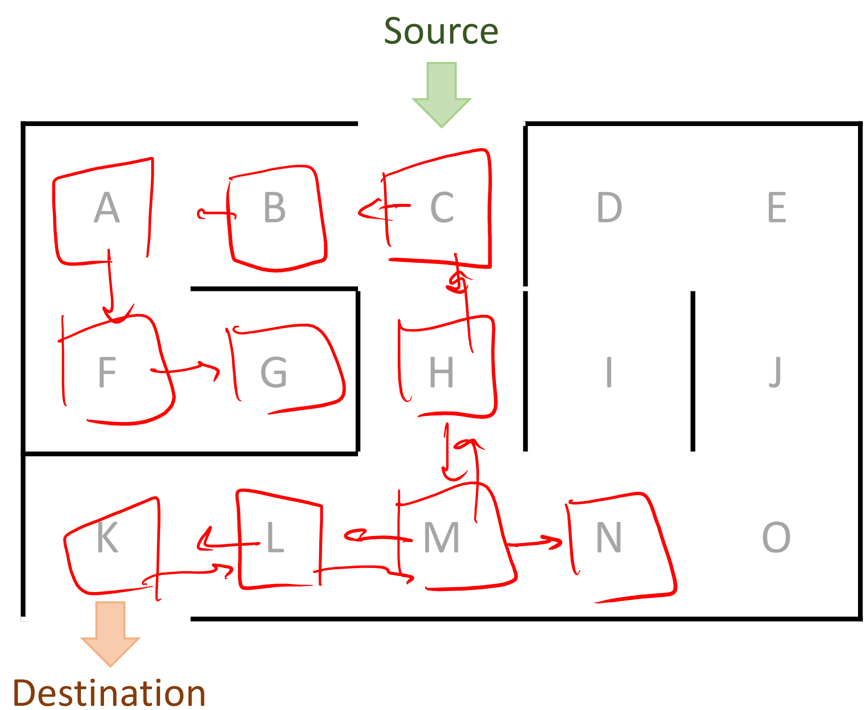
Now let's try using a **queue!**

Idea: Repeatedly explore and keep track of adjacent vertices.

Mark each vertex we visit, so we don't process each more than once.

Store as additional variable in vertices

BFS!



~~C B H A M F~~

~~L~~ N G K

Breadth First Search (BFS)

Breadth First Search (**BFS**):

Explore neighbors first, before moving to the next level of neighbors.

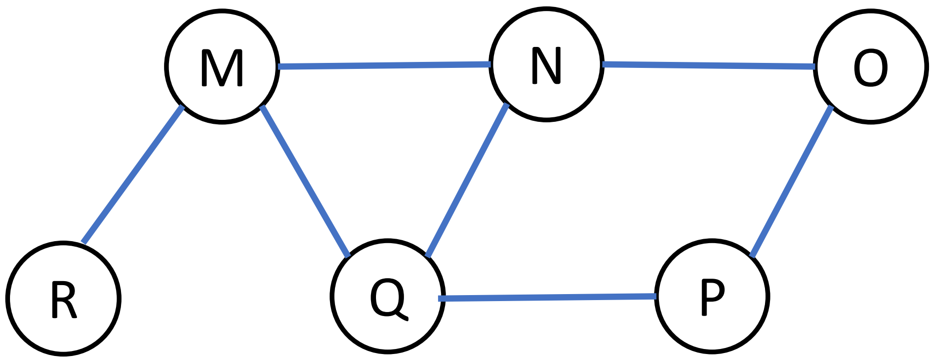
Repeatedly explore adjacent vertices using a queue
Mark each vertex we visit, so we don't process each more than once.

Example pseudocode:

```
BFS(Node start) {
  initialize queue q and enqueue start
  mark start as visited
  while(q is not empty) {
    next = q.dequeue() // and "process"
    for each node u adjacent to next
     if(u is not marked)
       mark u and enqueue onto q
  }
}
```

Practice time!

What is one possible order of visiting the nodes of the following graph when using Breadth First Search (BFS)?

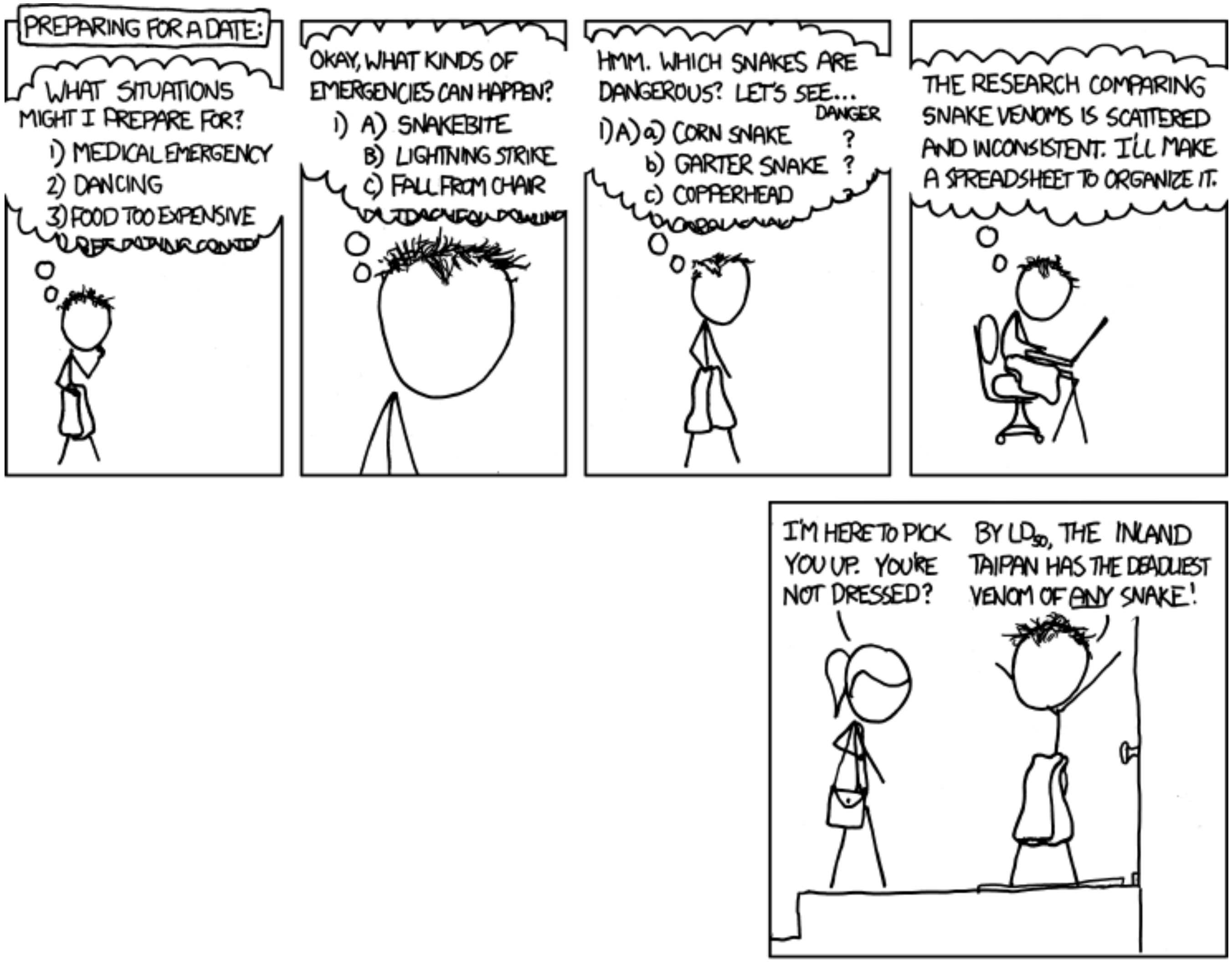


A) MNOPQR

B) NQMPOR

C) QMNPRO

D) QMNPOR



I REALLY NEED TO STOP USING DEPTH-FIRST SEARCHES.