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:

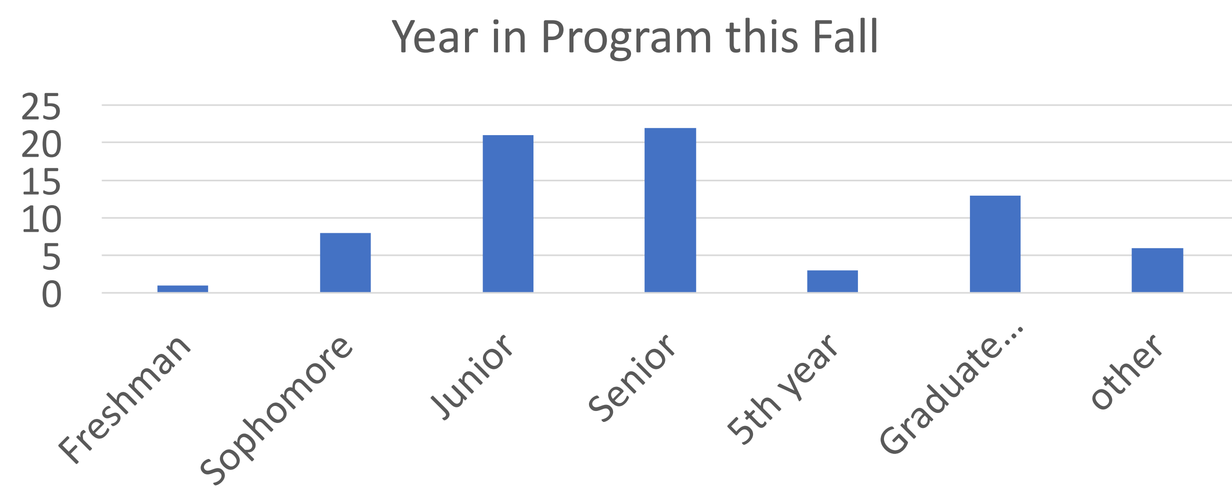
- Amount to write on printed slides
- Why using polling system for in-class exercises
- Concern about not getting through entire slide deck

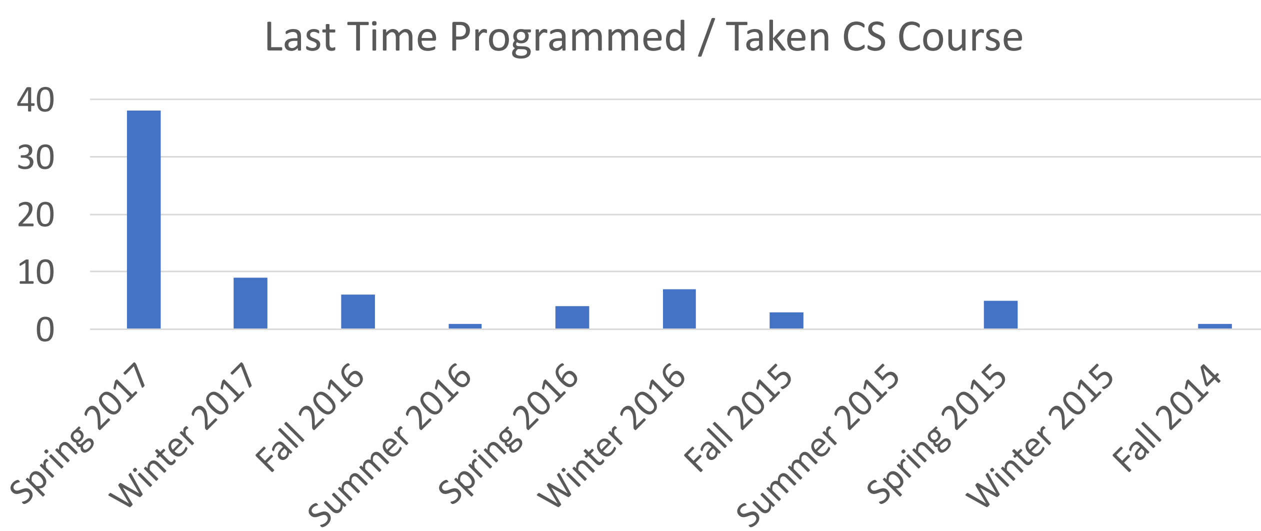
Wide range of student backgrounds!

Hence, using a range of teaching styles, pauses, etc.

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”

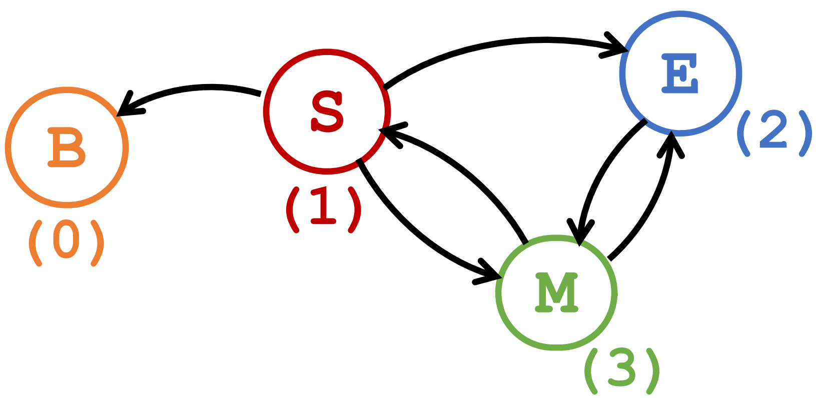
- But we need a data structure that represents graphs

- 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`

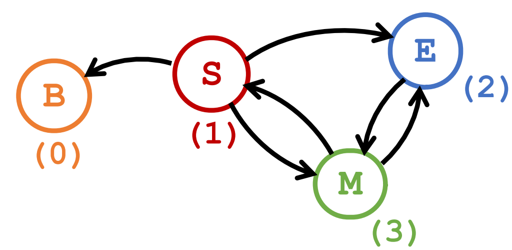


	0	1	2	3
0				
1				
2				
3				

Adjacency Matrix Properties

- Running time to:
  - Get a vertex's out-edges:
  - Get a vertex's in-edges:
  - Decide if some edge exists:
  - Insert an edge:
  - Delete an edge:
- Space requirements:
- Best for sparse or dense graphs?

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



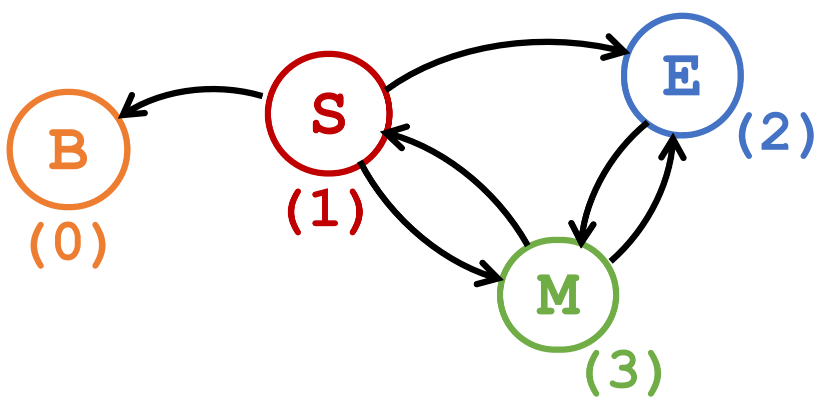
Adjacency Matrix Properties

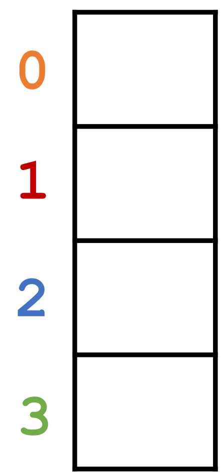
- How will the adjacency matrix vary for an *undirected graph*?
  - Undirected will be symmetric around the diagonal

- 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

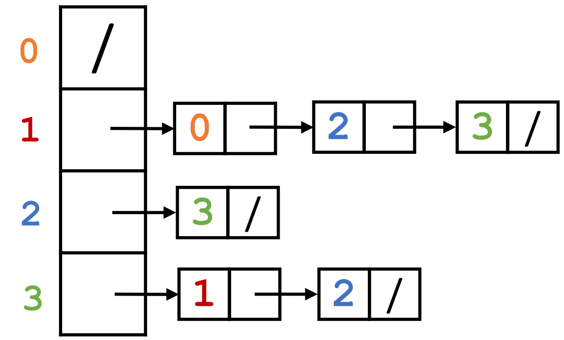
Adjacency List

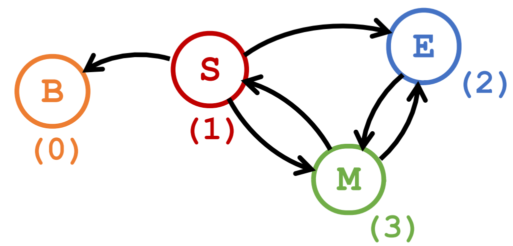
- Assign each node a number from `0` to `|V|-1`
- An array of length |V| in which each entry stores a list of all adjacent vertices (e.g., linked list)





Adjacency List Properties





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

Best for sparse or dense graphs?

Algorithms

Okay, we can represent graphs

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

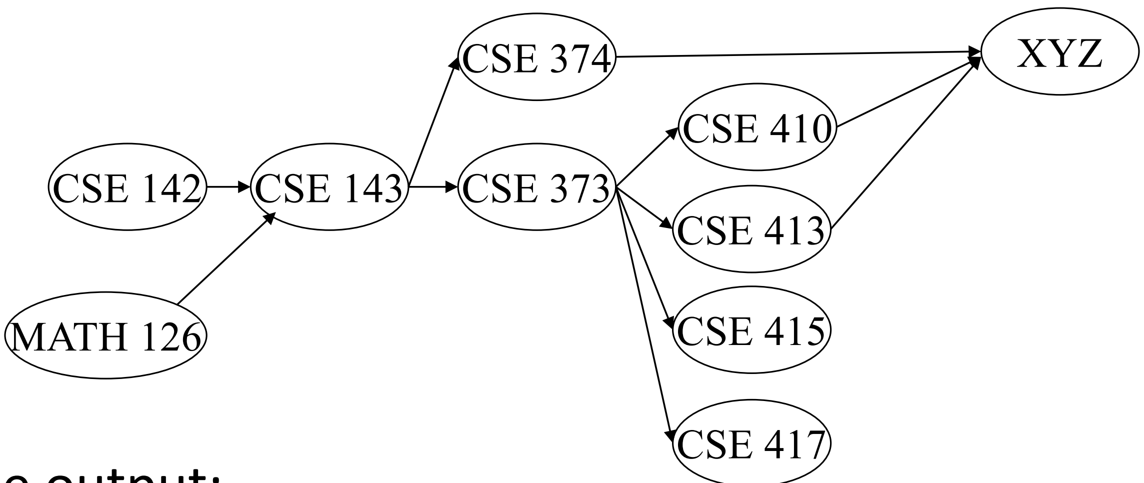
- Topological Sort
- Shortest Paths
  - Related: Determining if such a path exists
  - Depth First Search
  - Breadth First Search

Graphs: Topological Sort

Ordering vertices in a DAG

Topological Sort

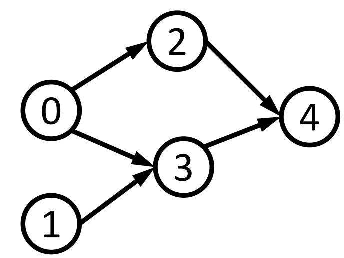
Topological sort: Given a DAG, order all the vertices so that every vertex comes before all of its neighbors



One example output:

Questions and comments

- Why do we perform topological sorts only on DAGs?



- Is there always a unique answer?

- Do some DAGs have exactly 1 answer?

- 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

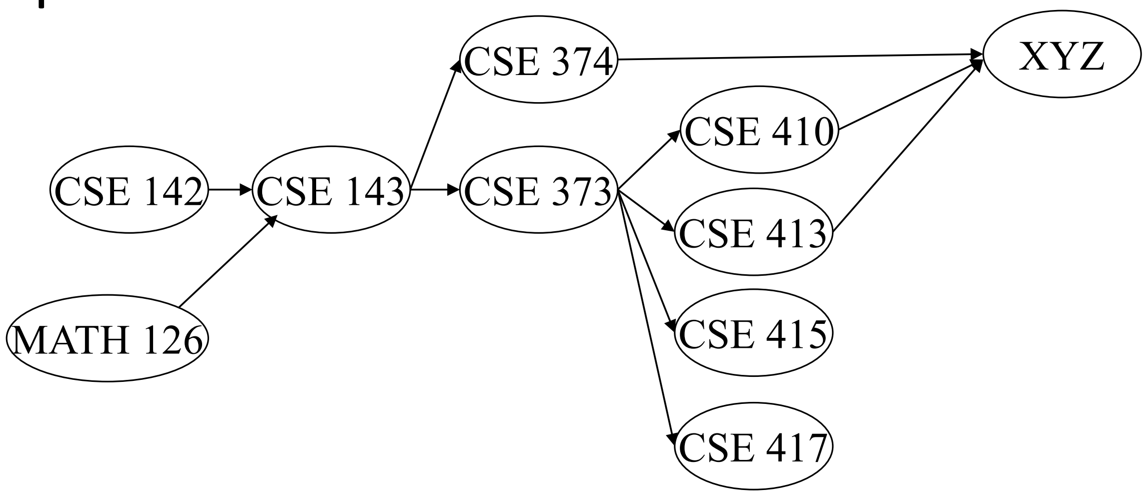
A First Algorithm for Topological Sort

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

Output:



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

Notice

- Needed a vertex with in-degree 0 to start
    - Will always have at least 1 because
- 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--;
}
```

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

Doing better

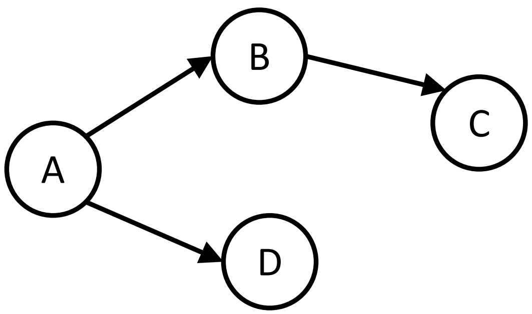
The trick is to avoid searching for a zero-degree node every time!

- 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?				
In-degree	0	1	1	2

Queue:

Output:

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);
   }
 }
```

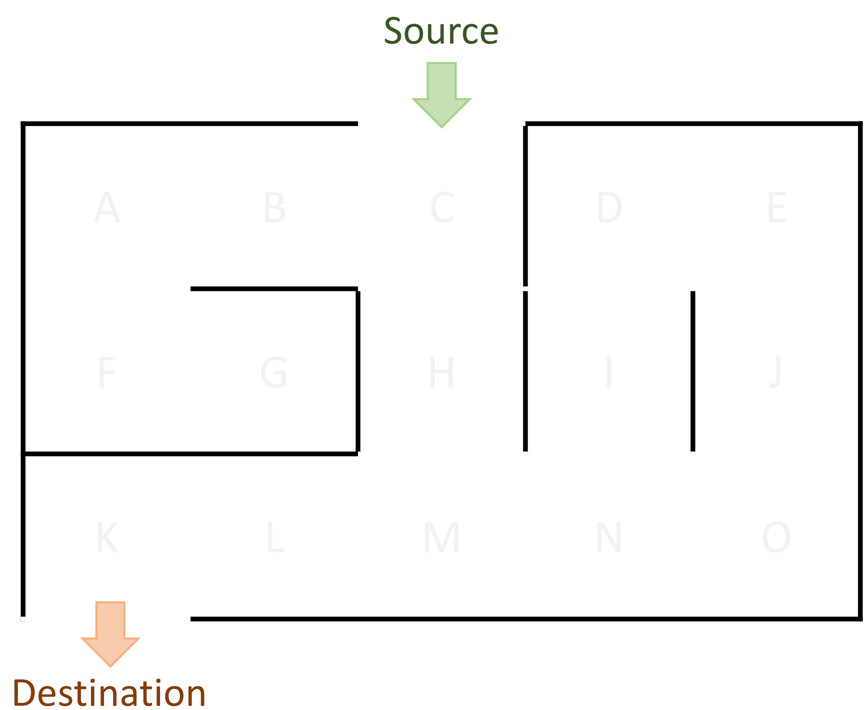
- What is the worst-case running time?
  - Initialization: (assuming adjacency list)
  - Sum of all enqueues and dequeues:
  - Sum of all decrements: (assuming adjacency list)
  - Total: – much better for sparse graph!

Graph Traversals

Depth- and Breadth- First Searches!

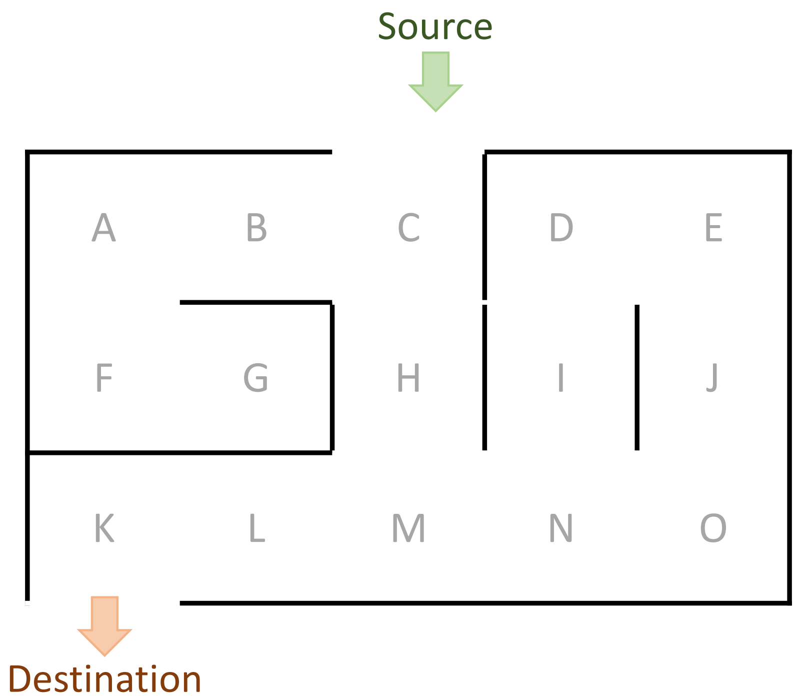
Introductory Example: Graph Traversals

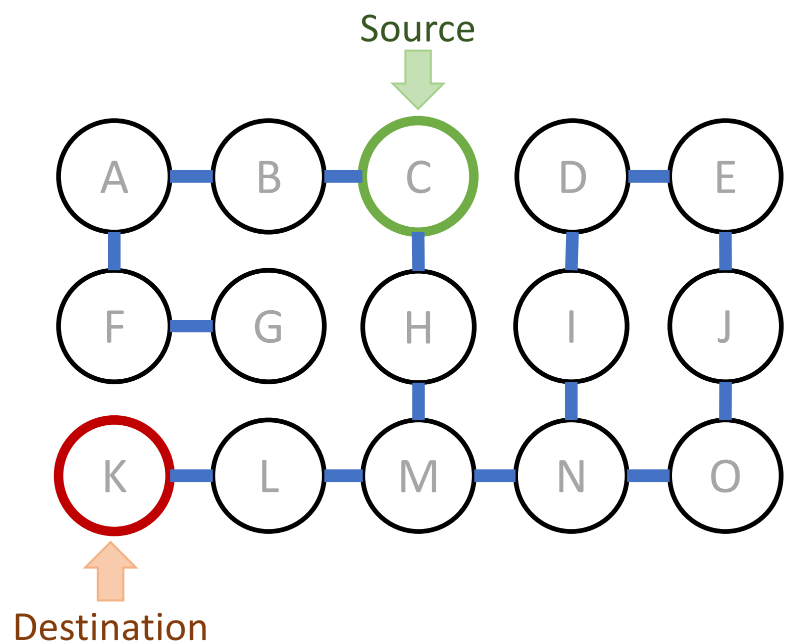
How would a computer systematically find a path through the maze?



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.





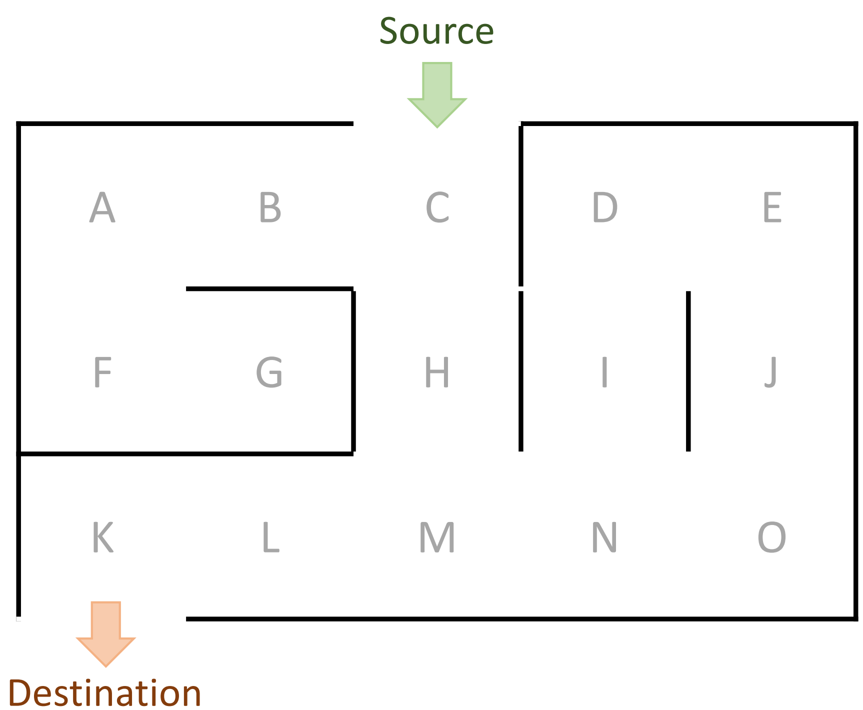
Find a path (if any) from one vertex to another.

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.

Store as additional variable in vertices



Depth First Search (DFS)

Depth First Search (**DFS**):

Explore as far as possible along each branch before backtracking

Repeatedly explore adjacent vertices using or

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

Example pseudocode:

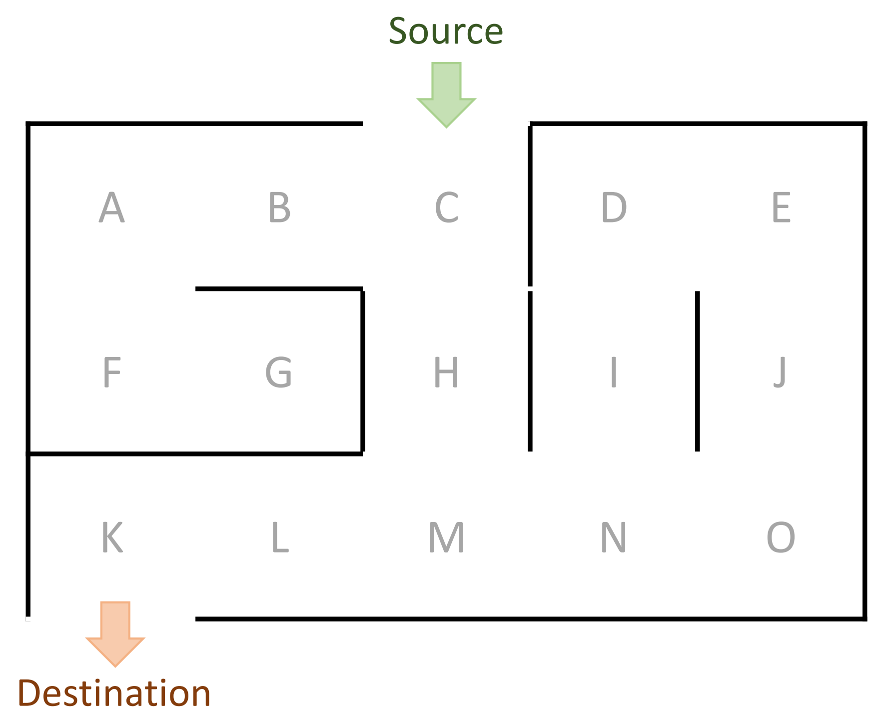
```
DFS(Node start) {
  mark and process start
  for each node u adjacent to start
    if u is not marked
      DFS(u)
}
```

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



Breadth First Search (BFS)

Breadth First Search (**BFS**):

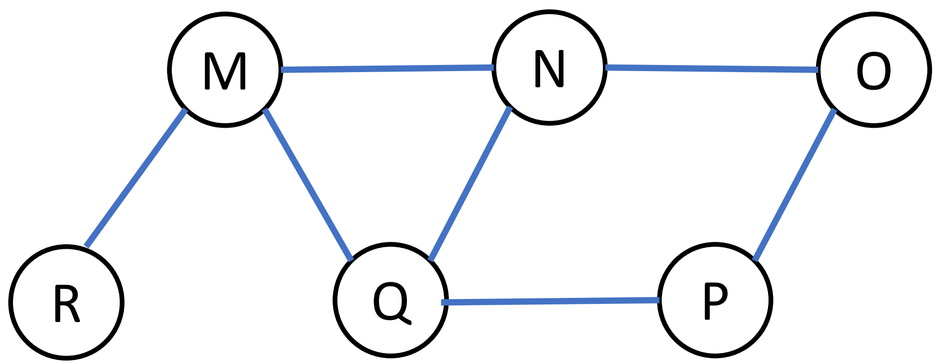
Explore neighbors first, before moving to the next level of neighbors.

Repeatedly explore adjacent vertices using
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

(space for scratch work / notes)

Running Time and Traversal Order

- Assuming `add` and `remove` are $O(1)$, entire traversal is
  - Use an adjacency list representation

- The order we traverse depends entirely on `add` and `remove`
  - For DFS:
  - For BFS:

Comparison (useful for Design Decisions!)

- Which one finds **shortest** paths?
  - i.e. which is better for "what is the shortest path from **x** to **y**" when there's more than one possible path?

- Which one can use less space in finding a path?

- A third approach:
  - *Iterative deepening (IDFS)*:
    - Try DFS but disallow recursion more than `K` levels deep
    - If that fails, increment `K` and start the entire search over
  - Like BFS, finds shortest paths. Like DFS, less space.

Graph Traversal Uses

In addition to finding paths, we can use graph traversals to answer:

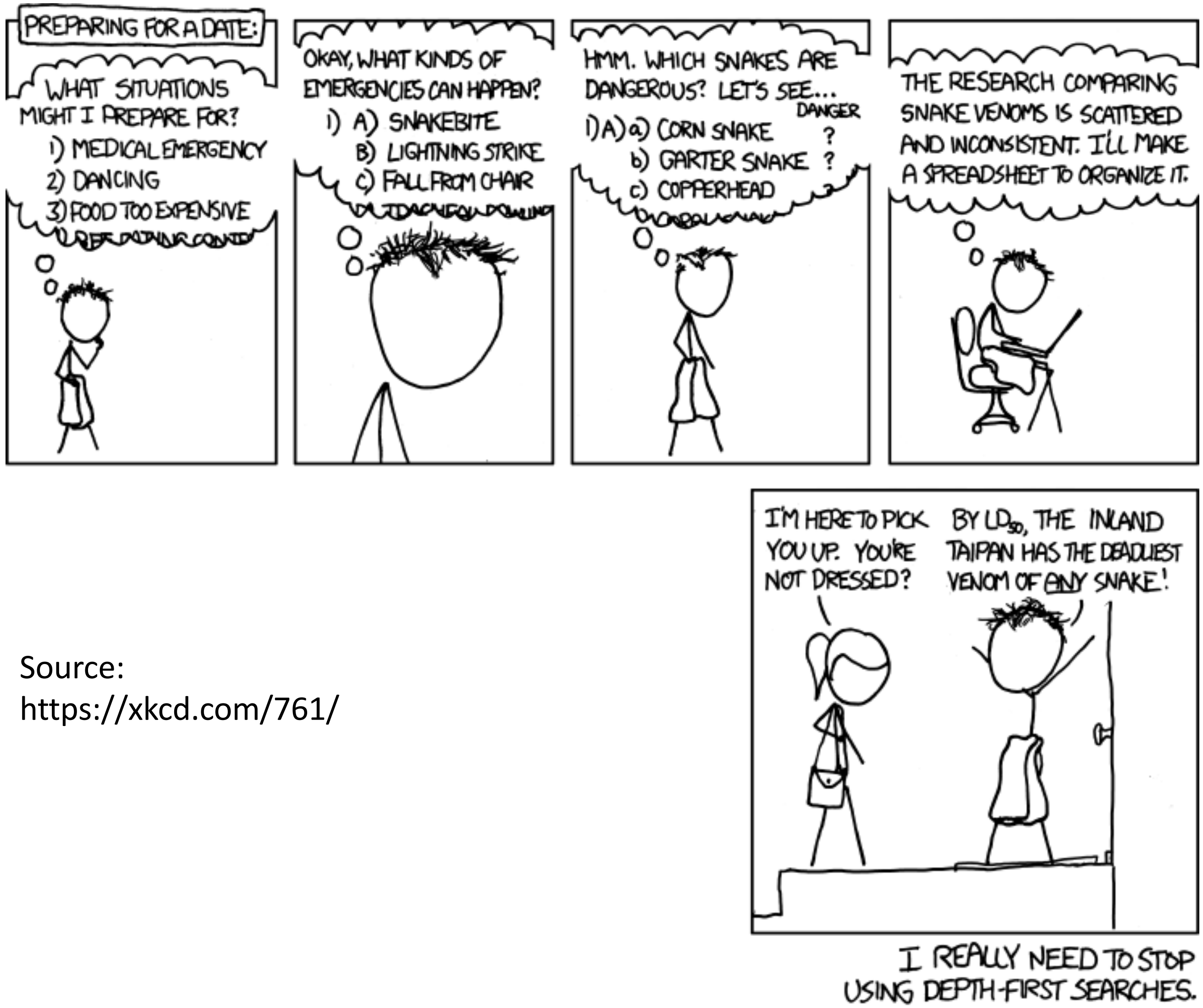
- What are all the vertices *reachable* from a starting vertex?
- Is an undirected graph connected?
- Is a directed graph strongly connected?

- But what if we want to actually output the path?

- How to do it:
  - Instead of just "marking" a node, store the previous node along the path
  - When you reach the goal, follow **`path`** fields back to where you started (and then reverse the answer)
  - If just wanted path *length*, could put the integer distance at each node instead once

Saving the Path

- Our graph traversals can answer the reachability question:
  - "Is there a path from node x to node y?"
- But what if we want to actually output the path?
- How to do it:
  - Instead of just "marking" a node, store the previous node along the path
  - When you reach the goal, follow **`path`** fields back to where you started (and then reverse the answer)
  - If just wanted path *length*, could put the integer distance at each node instead



Source:
https://xkcd.com/761/