Lecture 9: Finishing Dynamic Programming

Tim LaRock

larock.t@northeastern.edu

bit.ly/cs3000syllabus

Business

- Homework 2 due tonight at midnight Boston time
  - Solutions will be released 8AM Weds; absolutely no late submission without prior permission!
  - Submission now on Gradescope (see Piazza for info)
- Midterm 1 Wednesday 8PM through Friday 8PM
  - Questions answered during lecture Weds afternoon

Homework 2 Reminders

- Question 3: Assume you have a function IsMinimumLength() that tells you whether a valid chain C is minimum length in constant time

- Question 4 adjusted to be a bit easier
  - Part a: Write a recurrence for Opt(i,j)
  - Part b: Describe how to fill a dynamic programming table for Opt
  - Part c: Write in pseudocode how to fill the table

This week

Today:
- Find a dynamic programming solution for Edit Distance
- Wrap up dynamic programming
- Introduce basic features of graphs to get us started on graph algorithms

Tomorrow:
- First half-ish: Continue with graph algorithms
- Second half-ish: Answers to student-submitted questions
  - Form link sent out on Piazza

Thursday:
- No class while midterm exam is out

Edit Distance

Last week, we found a recurrence for Edit Distance:

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

…but we still need to develop a dynamic programming solution!

Edit Distance Recap

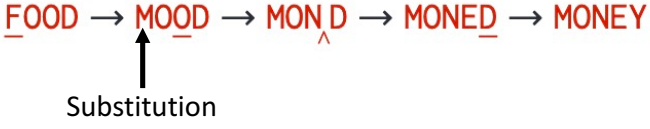
The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

Edit Distance Recap

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

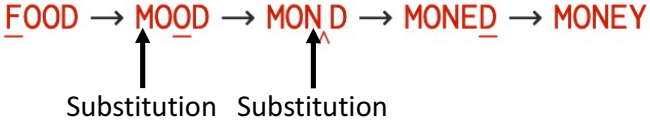
Edit Distance Recap

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.



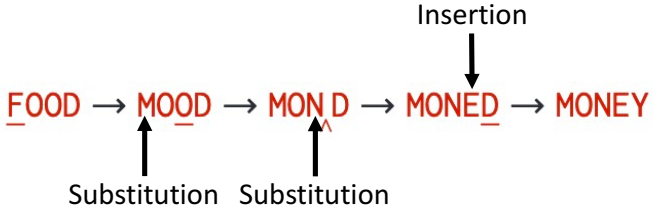
Edit Distance Recap

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.



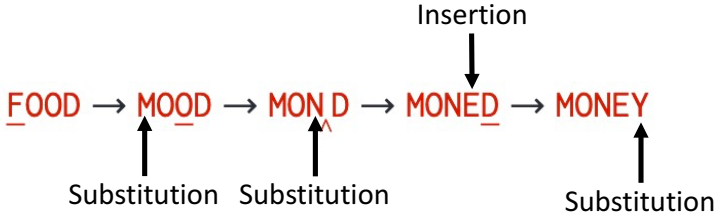
Edit Distance Recap

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.



Edit Distance Recap

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.



Edit Distance Recap

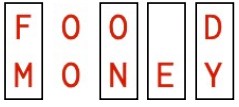
The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

```
F O O   D
M O N E Y
```

Alternative: Align the strings and count the differences

Edit Distance Recap

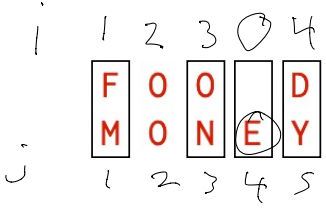
The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.



Alternative: Align the strings and count the differences

Edit Distance Recap

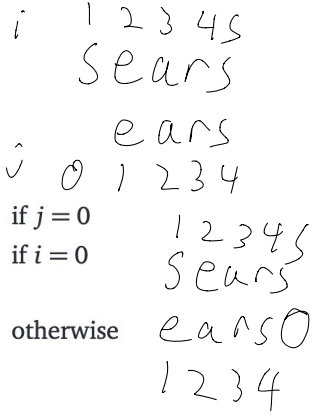
The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.



$EditDistance(\text{food}, \text{money}) = 4$

Alternative: Align the strings and count the differences

Formulating a recursive edit distance



Each call to $Edit(i, j)$makes a decision about how to align the last column in the substring.
There are three possibilities:

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

Arbitrary Case **Base Case**

1. Insertion

$Edit(i, j-1) + 1$ $Edit(0, j) = j$ to insert gaps in $A[0..j]$

---

2. Deletion

$Edit(i-1, j) + 1$ $Edit(i, 0) = i$ to delete characters from $B[0..i]$

---

3. Substitution

$Edit(i-1, j-1) + 1$, if $A[i] \neq B[j]$

$Edit(i-1, j-1)$, if $A[i] = B[j]$

Formulating a recursive edit distance

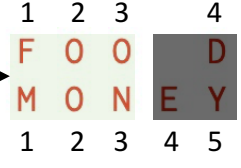
The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

For any two input strings $A[1..n]$ and $B[1..m]$, let

$$Edit(i, j)$$

denote the edit distance between prefixes $A[1..i]$ and $B[1..j]$. We need to compute $Edit(n, m)$.

Decisions here do not depend on what was already computed!



What should our subproblems be?

- Imagine that we have this alignment representation for the optimal edit distance
- Remove the last column
- What must be true of the remaining prefixes?
  - They must also be optimal!

Edit Distance Recap

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

Edit Distance Recap

The *edit distance* between two strings is the minimum number of insertions, deletions, and substitutions that will transform one string into the other.

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

0 if $A[i] = B[j]$ (no substitution is necessary!)
1 if $\mathrm{A[i]} \neq B[j]$

Edit Distance Dynamic Programming

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

$A[1 \ldots n]$

$B[1 \ldots m]$

We have our subproblems, so what data structure can we use for memoization?

Edit Distance Dynamic Programming

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

We have our subproblems, so what data structure can we use for memoization?

We can use a two dimensional array:

$Edit[0..n, 0..m]$

Edit Distance Dynamic Programming

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?

Edit Distance Dynamic Programming

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?

$Edit[i,j]$ potentially depends on three adjacent entries:

Edit Distance Dynamic Programming

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{Bmatrix} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{Bmatrix} & \text{otherwise} \end{cases}$$

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?

$Edit[i,j]$ potentially depends on three adjacent entries:
$Edit[i, j-1]$

Edit Distance Dynamic Programming

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?

$Edit[i,j]$ potentially depends on three adjacent entries:

$Edit[i, j-1]$

$Edit[i-1, j]$

Edit Distance Dynamic Programming

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?

$Edit[i,j]$ potentially depends on three adjacent entries:

$Edit[i, j-1]$

$Edit[i-1, j]$

$Edit[i-1, j-1]$

Edit Distance Dynamic Programming

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{Bmatrix} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{Bmatrix} & \text{otherwise} \end{cases}$$

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?

$Edit[i,j]$ potentially depends on three adjacent entries:

$Edit[i,j-1]$

$Edit[i-1,j]$

$Edit[i-1,j-1]$

We can fill it from top ($i = j = 0$) to bottom ($i = n, j = m$).

Edit Distance Dynamic Programming

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?
Top ($i = j = 0$) to bottom ($i = n, j = m$).

Edit Distance Dynamic Programming

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?
Top ($i = j = 0$) to bottom ($i = n, j = m$).

```
Edit(A[1..n], B[1..m]):
  for j ← 0 to m:
    S[0,j] ← j
  for i ← 1 to n:
    S[i,0] ← i
    for j ← 1 to m:
      insert ← Edit[i,j−1] + 1
      delete ← Edit[i−1,j] + 1

      if A[i] = B[j]:
        sub ← Edit[i−1,j−1]
      else:
        sub ← Edit[i−1,j−1] + 1
      Edit[i,j] = min(insert, delete, sub)
  return Edit[n,m]
```

Edit Distance Example

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?
Top ($i = j = 0$) to bottom ($i = n, j = m$).

```
Edit(A[1..n], B[1..m]):
  for j ← 0 to m:
    S[0,j] ← j
  for i ← 1 to n:
    S[i,0] ← i
    for j ← 1 to m:
      insert ← Edit[i,j−1] + 1
      delete ← Edit[i−1,j] + 1

      if A[i] = B[j]:
        sub ← Edit[i−1,j−1]
      else:
        sub ← Edit[i−1,j−1] + 1
      Edit[i,j] = min(insert, delete, sub)
  return Edit[n,m]
```

Edit Distance Example

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

```
Edit(A[1..n], B[1..m]):
  for j ← 0 to m:
    S[0,j] ← j
  for i ← 1 to n:
    S[i,0] ← i
    for j ← 1 to m:
      insert ← Edit[i,j−1] + 1
      delete ← Edit[i−1,j] + 1

      if A[i] = B[j]:
        sub ← Edit[i−1,j−1]
      else:
        sub ← Edit[i−1,j−1] + 1
      Edit[i,j] = min(insert, delete, sub)
  return Edit[n,m]
```

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?
Top ($i = j = 0$) to bottom ($i = n, j = m$).

f	o	o		d
m	o	n	e	y

$Edit[i,j]$

	0	1	2	3	4	5
0						
1						
2						
3						
4						

Edit Distance Example

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

```
Edit(A[1..n], B[1..m]):
  for j ← 0 to m:
    S[0,j] ← j
  for i ← 1 to n:
    S[i,0] ← i
    for j ← 1 to m:
      insert ← Edit[i,j−1] + 1
      delete ← Edit[i−1,j] + 1

      if A[i] = B[j]:
        sub ← Edit[i−1,j−1]
      else:
        sub ← Edit[i−1,j−1] + 1
      Edit[i,j] = min(insert, delete, sub)
  return Edit[n,m]
```

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?
Top ($i = j = 0$) to bottom ($i = n, j = m$).

	f	o	o		d
m	o	n	e	y	

	$j=0$					
$i=0$	0	1	2	3	4	5
$i=1$	1	1	2	3	4	5
$i=2$	2	2	1	2	3	4
$i=3$	3	3	2	2	3	4
$i=4$	4	4	3	3	3	4

Edit Distance

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

```
Edit(A[1..n], B[1..m]):
  for j ← 0 to m:
    S[0,j] ← j
  for i ← 1 to n:
    S[i,0] ← i
    for j ← 1 to m:
      insert ← Edit[i,j−1] + 1
      delete ← Edit[i−1,j] + 1

      if A[i] = B[j]:
        sub ← Edit[i−1,j−1]
      else:
        sub ← Edit[i−1,j−1] + 1
      Edit[i,j] = min(insert, delete, sub)
  return Edit[n,m]
```

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?
Top ($i = j = 0$) to bottom ($i = n, j = m$).

Finally: What is the running time and space requirement of this algorithm?

Edit Distance

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

```
Edit(A[1..n], B[1..m]):
  for j ← 0 to m:
    S[0,j] ← j
  for i ← 1 to n:
    S[i,0] ← i
    for j ← 1 to m:
      insert ← Edit[i,j−1] + 1
      delete ← Edit[i−1,j] + 1

      if A[i] = B[j]:
        sub ← Edit[i−1,j−1]
      else:
        sub ← Edit[i−1,j−1] + 1
      Edit[i,j] = min(insert, delete, sub)
  return Edit[n,m]
```

$2m$

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?
Top ($i = j = 0$) to bottom ($i = n, j = m$).

Finally: What is the running time and space requirement of this algorithm?

$$O(nm)$$

Edit Distance Wrap

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{cases} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{cases} & \text{otherwise} \end{cases}$$

Data Structure?
Two dimensional array:
$Edit[0..n, 0..m]$

What order should we fill the array in?
Top ($i = j = 0$) to bottom ($i = n, j = m$).

Running time and space requirements?
$O(nm)$

```
Edit(A[1..n], B[1..m]):
  for j ← 0 to m:
    S[0,j] ← j
  for i ← 1 to n:
    S[i,0] ← i
    for j ← 1 to m:
      insert ← Edit[i,j−1] + 1
      delete ← Edit[i−1,j] + 1

      if A[i] = B[j]:
        sub ← Edit[i−1,j−1]
      else:
        sub ← Edit[i−1,j−1] + 1
      Edit[i,j] = min(insert, delete, sub)
  return Edit[n,m]
```

As usual, we:

- Specified our problem in terms of solutions to smaller subproblems
- Found a data structure to store solutions to the subproblems
- Filled the data structure in a smart way to avoid recomputing any solutions
- The running time and space requirements were immediately clear from the way we filled the data structure

Clarification: Top down Vs. Bottom up

Consider the following two statements:

1. To ace the midterm exam, I will practice some divide and conquer problems on the homework, and to solve the practice problems I first need to attend the lecture to learn about divide and conquer.
2. I will attend the lecture to learn about divide and conquer, then I will practice some problems on the homework, then I will ace the midterm exam.

They say the same thing, but the first is “top down” and the second is “bottom up”!

Clarification: Top down Vs. Bottom up

In algorithms:

- Top down is memoization
  - We discussed for Fibonacci numbers
  - In this case, your solution still ”looks recursive”, but every call either fills an entry in the table or uses the table to get a next solution
- Bottom up is also called “tabulation”
  - It usually corresponds to starting at the base case and building every subsequent solution from there.

Observation: A tradeoff between the two is that tabulation is often conceptually simpler to implement, but memoization is easier to think about if you already have a full recursive solution.

Key point: The result on a given input should be equivalent for our purposes!

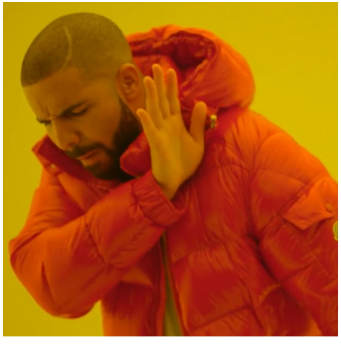
Dynamic Programming Wrap

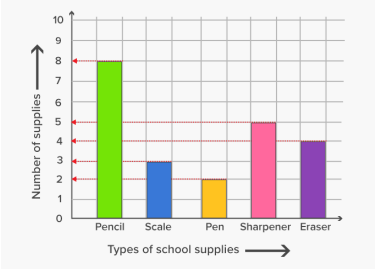
- Recursion is often convenient for solving problems, but it can result in a lot of redundant computation that makes recursive algorithms very slow
- Dynamic programming is a technique to "transform" recursive problems in to iterative problems
  - Define subproblems recursively
  - Determine a suitable data structure (we only studied arrays so far, but others are possible)
  - Determine the dependencies between subproblems, which suggests the order in which they should be evaluated
    - Order may be top-down (memoize) or bottom-up (tabulate)
  - Write down the pseudocode for the algorithm and evaluate runtime/space requirements
- Dynamic programming is not always faster for all inputs!
  - It might compute a solution to every possible subproblem, even those that are not directly used to compute the output!
- Dynamic programming is more than filling tables (see discussion in Erickson)
  - But in practice it *is* largely about filling tables

Graphs

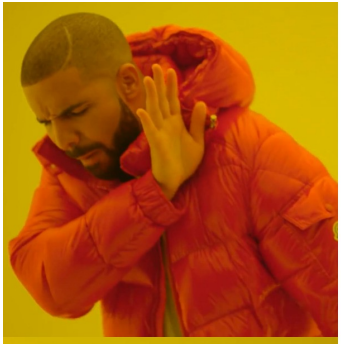
Graphs: what are they?

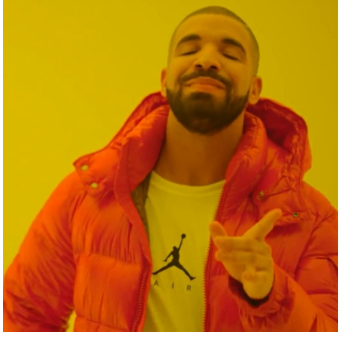
Graphs: what are they?

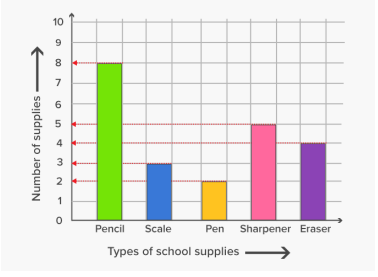


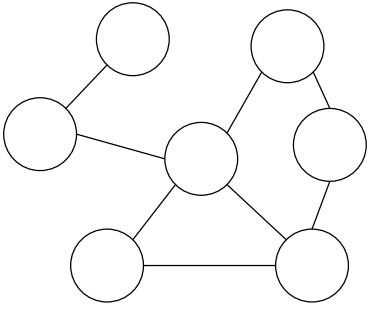


Graphs: what are they?









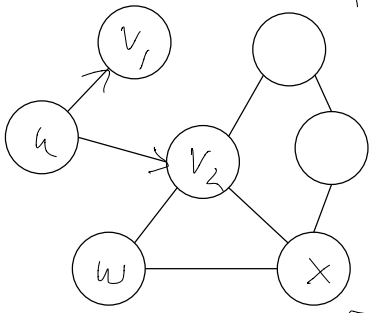
Graphs

V set of vertices
E set of edges

A *graph* $G = (V, E)$ consists of
- *vertices* $v \in V$ and
- *edges* $e \in E$, indicating with two vertices u, v are *connected*

A graph is:
- *directed* if the vertices in each edge are *ordered*
  - If we are being precise, we say "the edge *from* u *to* v"
  - $(u, v) \in E \not\Rightarrow (v, u) \in E$
- *undirected* if its edges are not ordered.
  - "the edge *between* u *and* v"
  - $(u, v) \in E \Rightarrow (v, u) \in E$

(u, v_1) (u, v_2) ~~(v_2, u)~~ ~~(v_1, u)~~



$(w, x) \in E$
$(x, w) \in E$

Graphs “vs” Networks

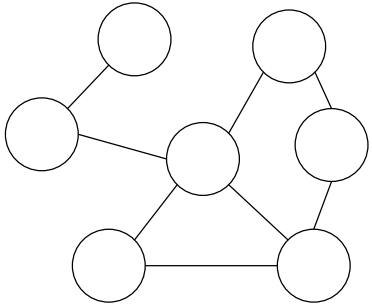
Graphs and Networks are often used interchangeably

A graph is a mathematical (topological) object defined on the previous slide

A network is a concept referring to “things relating to other things” and networks are almost always studied and represented as graphs

“Networks are graphs with meaning”

We will almost always study graphs in this class since their interpretation is often not important to the algorithms we are developing



I study networks for a living

Northeastern University
Network Science Institute

People

Timothy LaRock

Network Science PhD Candidate

More on my website and: https://www.networkscienceinstitute.org/

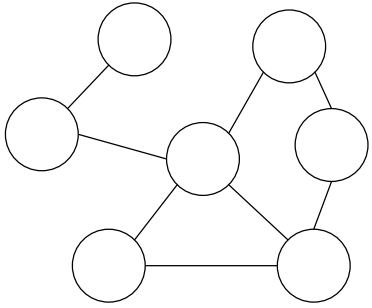
Graphs

Vertices → nodes
edges → links

A *graph* $G = (V, E)$ consists of
- *vertices* $v \in V$ and
- *edges* $e \in E$, indicating with two vertices u, v are *connected*

We will often refer to vertices as *nodes* and may also refer to edges as *links*. Consider these interchangeable!

A graph is:
- *directed* if the vertices in each edge are *ordered*
  - If we are being precise, we say "the edge *from* u *to* v"
  - $(u, v) \in E \not\Rightarrow (v, u) \in E$
- *undirected* if its edges are not ordered.
  - "the edge *between* u *and* v"
  - $(u, v) \in E \Rightarrow (v, u) \in E$

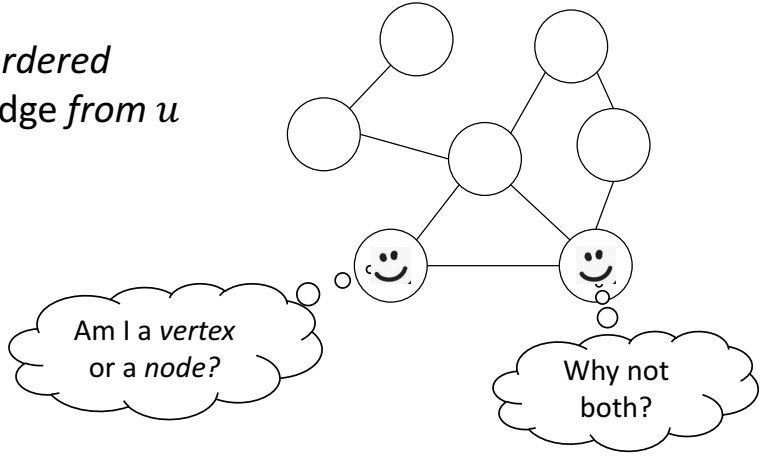


Graphs

A *graph* $G = (V, E)$ consists of
- *vertices* $v \in V$ and
- *edges* $e \in E$, indicating with two vertices u, v are *connected*

We will often refer to vertices as *nodes* and may also refer to edges as *links*. Consider these interchangeable!

A graph is:
- *directed* if the vertices in each edge are *ordered*
  - If we are being precise, we say "the edge *from* u *to* v"
  - $(u, v) \in E \nRightarrow (v, u) \in E$
- *undirected* if its edges are not ordered.
  - "the edge *between* u *and* v"
  - $(u, v) \in E \Rightarrow (v, u) \in E$



Paths through graphs

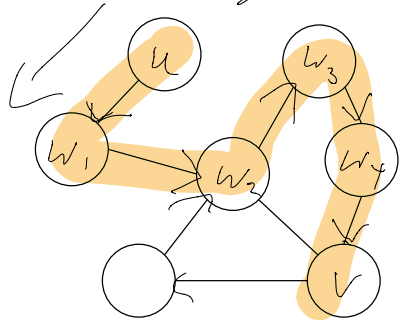
A *path* P from vertex u to vertex v through a graph is an ordered sequence of consectuive edges from E:

$$P = \{(u, w_1), (w_1, w_2), \dots, (w_{k-1}, v)\}$$

A *graph* $G = (V, E)$ is connected if for every pair of nodes u and v there is a *path* from u to v.

In a directed graph, there must be both a path from u to v and from v to u.

A graph with a path in one direction for all u to v is "weakly connected"



A graph w/ paths in both directions u to v is "strongly co

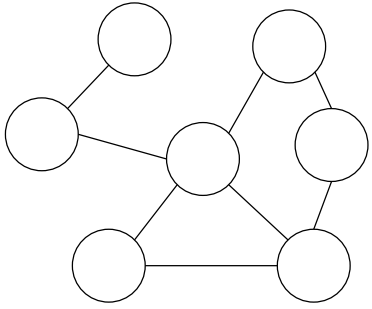
Much more to come on graphs!

Graph traversal

Finding shortest paths

Minimum spanning trees

Flow algorithms



This week

Tomorrow:

- First half-ish: Start graph algorithms in earnest
- Second half-ish: Answers to student-submitted questions
  - Form link sent out on Piazza

Thursday:

- No class while midterm exam is out

Next Monday:

- No class due to Memorial Day