Algorithms and Data Structures for Data Science

Hashing 2

CS 277

Brad Solomon

April 10, 2024

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

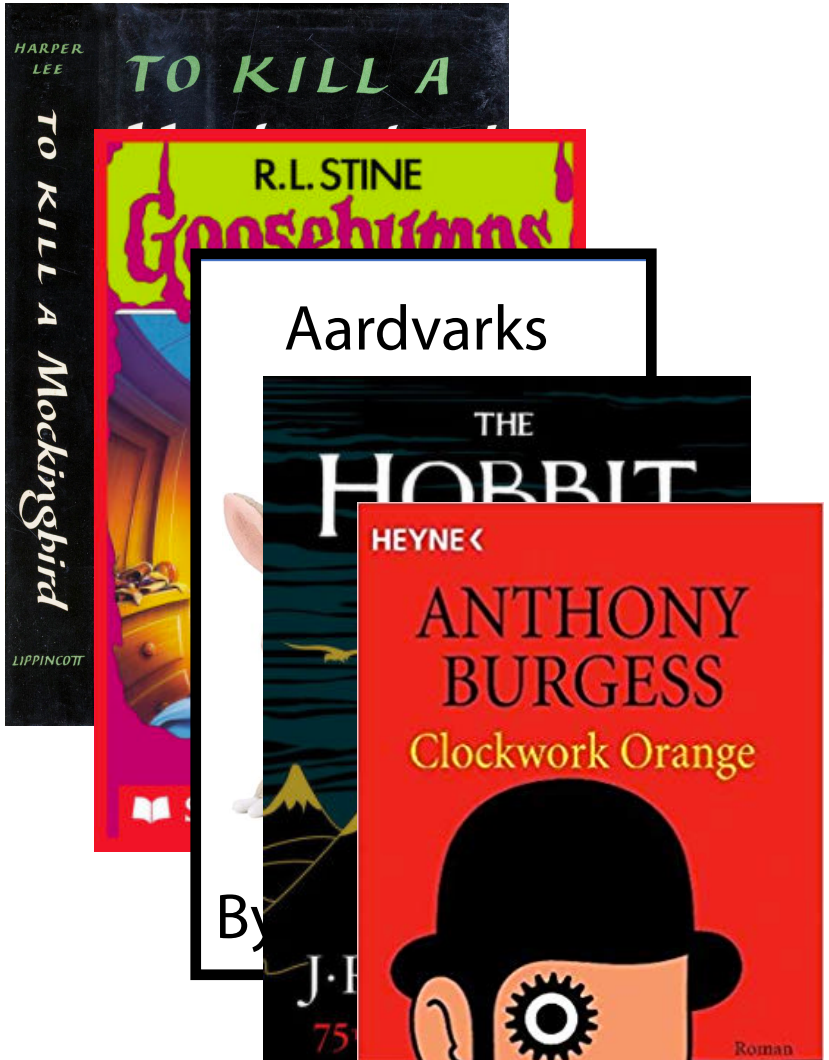
Review fundamentals of hash tables

Introduce closed hashing approaches to hash collisions

Determine when and how to resize a hash table

Data Structure Review

Data as key, value pairs is an extremely common format in data science



A Hash Table based Dictionary

```
1 d = {}
2 d[k] = v
3 print(d[k])
```

A **Hash Table** consists of three things:

1. A hash function

2. A data storage structure

3. A method of addressing *hash collisions*

Open vs Closed Hashing

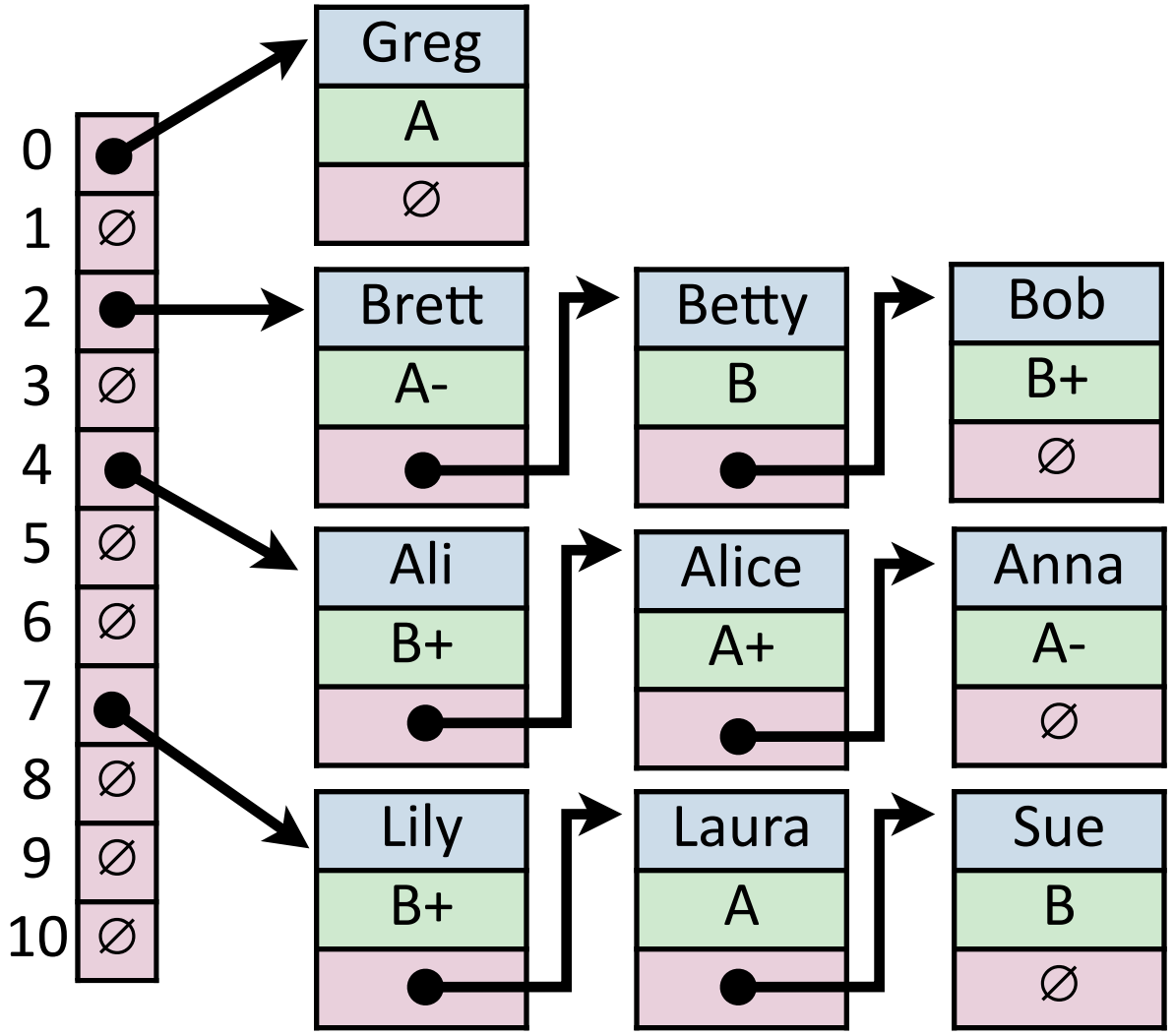
Addressing hash collisions depends on your storage structure.

- **Open Hashing:** store *k,v* pairs externally

- **Closed Hashing:** store *k,v* pairs in the hash table

Hash Table (Separate Chaining)

Key	Value	Hash
Bob	B+	2
Anna	A-	4
Alice	A+	4
Betty	B	2
Brett	A-	2
Greg	A	0
Sue	B	7
Ali	B+	4
Laura	A	7
Lily	B+	7



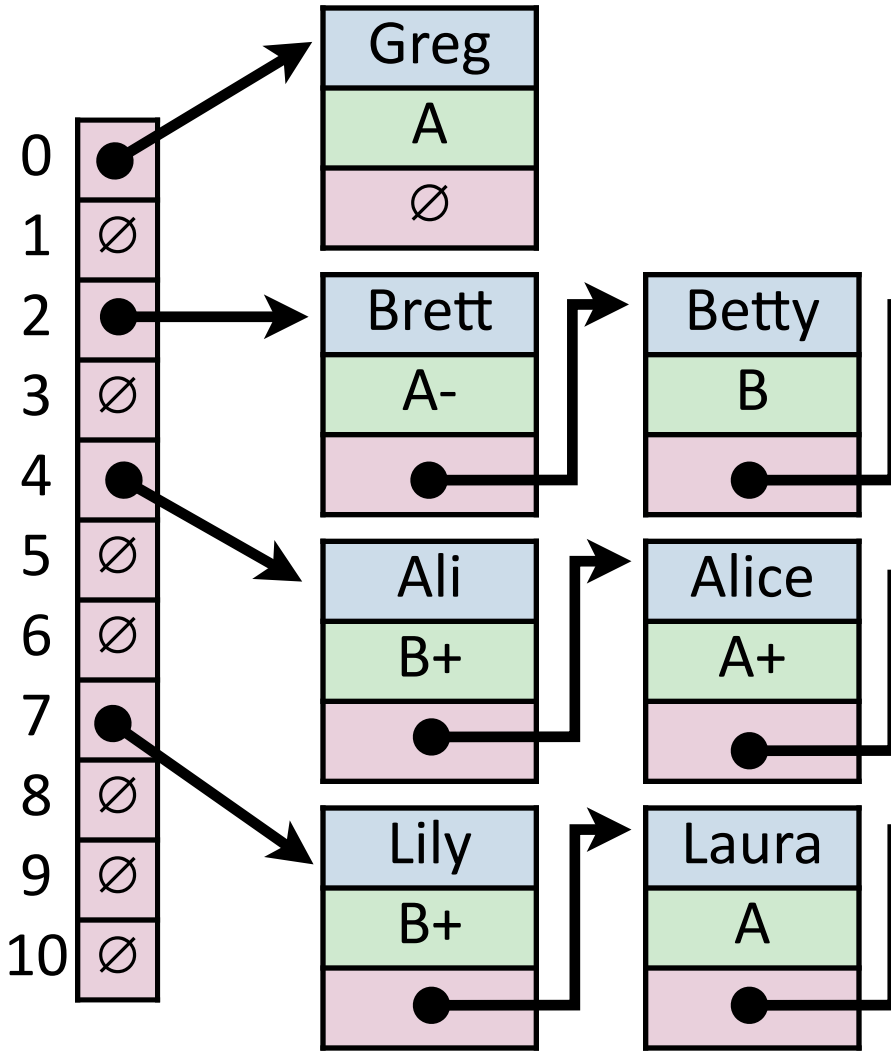
Hash Table (Separate Chaining)

For hash table of size *m* and *n* elements:

Find runs in: _____

Insert runs in: _____

Remove runs in: _____



Hash Table

Worst-Case behavior is bad — but what about randomness?

1) **Fix *h***, our hash, and assume it is good for ***all keys***:

Simple Uniform Hashing Assumption (SUHA)

Simple Uniform Hashing Assumption

Given table of size m, a simple uniform hash, h, implies

$$\forall k_1, k_2 \in U \text{ where } k_1 \neq k_2 \,,\; Pr(h[k_1] = h[k_2]) = \frac{1}{m}$$

Uniform: keys are equally likely to hash to any position

Independent: All keys hash independently of each other

Separate Chaining Under SUHA

Given table of size m and n inserted objects

Claim: Under SUHA, expected length of chain is $\frac{n}{m}$

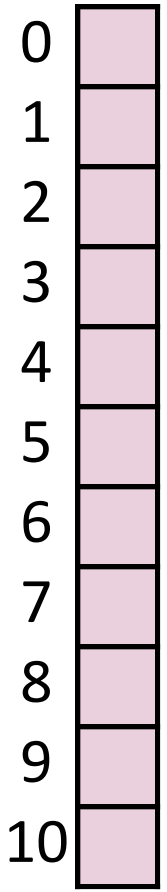
Separate Chaining Under SUHA

Under SUHA, a hash table of size *m* and *n* elements:

Find runs in: _____.

Insert runs in: _____.

Remove runs in: _____.



(Example of closed hashing)

Collision Handling: Probe-based Hashing

S = { 1, 8 , 15}

|S| = n

h(k) = k % 7

|Array| = m

(Example of closed hashing)

Collision Handling: Linear Probing

S = { 16, 8, 4, 13, 29, 11, 22 } **|S| = n**

h(k) = k % 7 **|Array| = m**

h(k, i) = (k + i) % 7

Try h(k) = (k + 0) % 7, if full…

Try h(k) = (k + 1) % 7, if full…

Try h(k) = (k + 2) % 7, if full…

Try …

(Example of closed hashing)

Collision Handling: Linear Probing

S = { 16, 8, 4, 13, 29, 11, 22 } **|S| = n**

h(k, i) = (k + i) % 7 **|Array| = m**

`_find(29)`

0	22
1	8
2	16
3	29
4	4
5	11
6	13

(Example of closed hashing)

Collision Handling: Linear Probing

S = { 16, 8, 4, 13, 29, 11, 22 }

|S| = n

h(k, i) = (k + i) % 7

|Array| = m

`_remove(16)`

0	22
1	8
2	16
3	29
4	4
5	11
6	13

A Problem w/ Linear Probing

Primary clustering:

0	
1	$\mathbf{1_1}$
2	$\mathbf{1_2}$
3	$\mathbf{3_1}$
4	$\mathbf{1_3}$
5	$\mathbf{3_2}$
6	
7	
8	
9	

Description:

Remedy:

(Example of closed hashing)

Collision Handling: Quadratic Probing

S = { 16, 8, 4, 13, 29, 12, 22 }

|S| = n

h(k) = k % 7

|Array| = m

0	
1	**8**
2	**16**
3	
4	**4**
5	
6	**13**

h(k, i) = (k + i*i) % 7

Try h(k) = (k + 0) % 7, if full…

Try h(k) = (k + 1*1) % 7, if full…

Try h(k) = (k + 2*2) % 7, if full…

Try …

A Problem w/ Quadratic Probing

Secondary clustering:

0	O_1
1	O_2
2	
3	
4	O_3
5	
6	
7	
8	
9	O_4

Description:

Remedy:

(Example of closed hashing)

Collision Handling: Double Hashing

S = { 16, 8, 4, 13, 29, 11, 22 }

$h_1(k) = k \% 7$

$h_2(k) = 5 - (k \% 5)$

|S| = n

|Array| = m

0	
1	**8**
2	**16**
3	
4	**4**
5	
6	**13**

$h(k, i) = (h_1(k) + i*h_2(k)) \% 7$

Try $h(k) = (k + 0*h_2(k)) \% 7$, if full…

Try $h(k) = (k + 1*h_2(k)) \% 7$, if full…

Try $h(k) = (k + 2*h_2(k)) \% 7$, if full…

Try …

Running Times

(Expectation under SUHA)

(Don't memorize these equations, no need.)

Open Hashing:

insert: _____.

find/ remove: _____.

Closed Hashing:

insert: _____.

find/ remove: _____.

Running Times *(Don't memorize these equations, no need.)*

The expected number of probes for find(key) under SUHA

Linear Probing:

- Successful: **½(1 + 1/(1-α))**
- Unsuccessful: **½(1 + 1/(1-α))²**

Double Hashing:

- Successful: **1/α * ln(1/(1-α))**
- Unsuccessful: **1/(1-α)**

Separate Chaining:

- Successful: **1 + α/2**
- Unsuccessful: **1 + α**

Instead, observe:

- As α increases:

- If α is constant:

Running Times

The expected number of probes for find(key) under SUHA

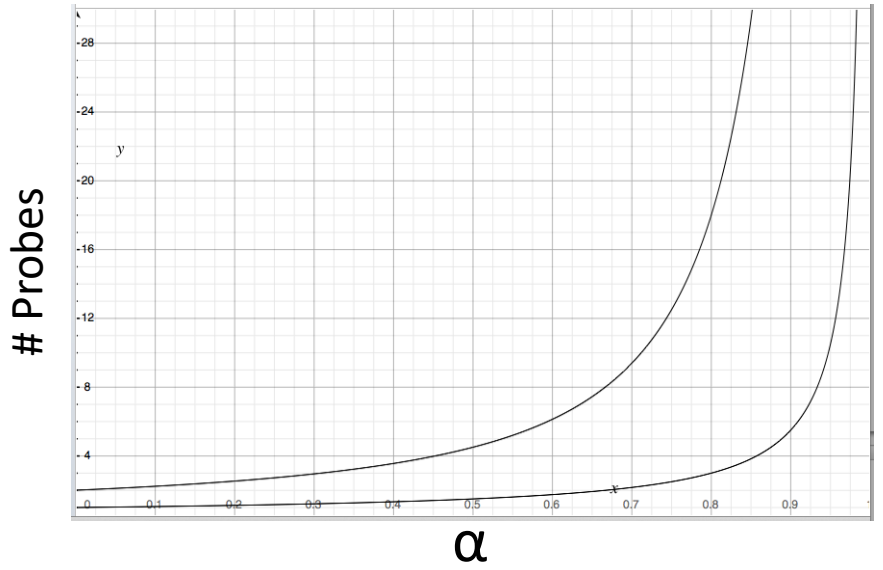
Linear Probing:

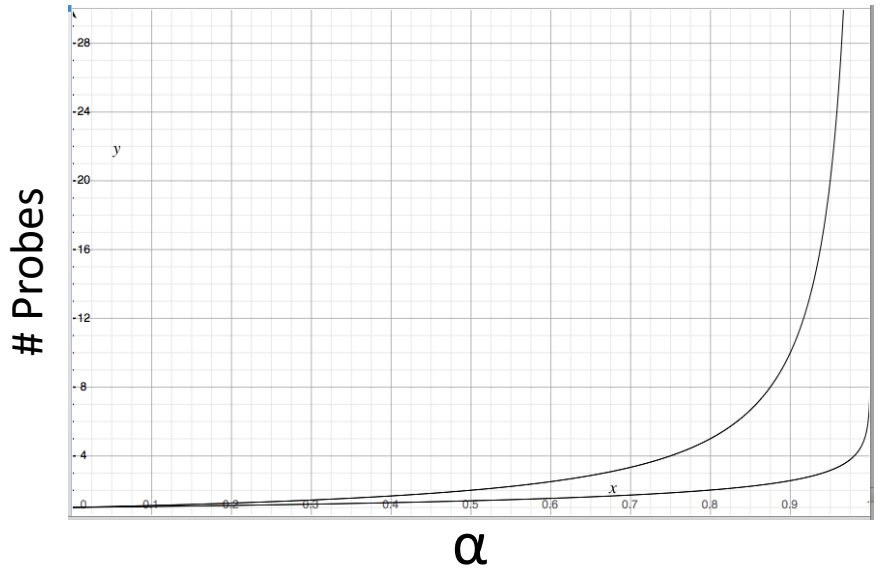
- Successful: **½(1 + 1/(1-α))**
- Unsuccessful: **½(1 + 1/(1-α))²**

Double Hashing:

- Successful: **1/α * ln(1/(1-α))**
- Unsuccessful: **1/(1-α)**

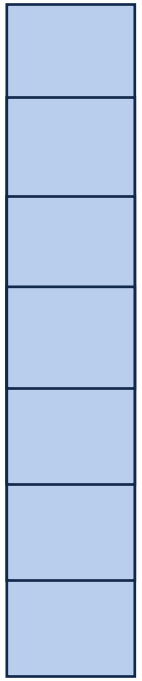
When do we resize?

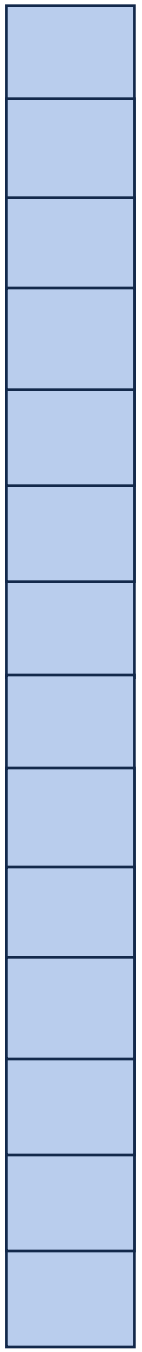




Resizing a hash table

How do you resize?





Which collision resolution strategy is better?

- Big Records:

- Structure Speed:

What structure do hash tables implement?

What constraint exists on hashing that doesn’t exist with BSTs?

Why talk about BSTs at all?

Running Times Review

	Hash Table	AVL	Linked List
Find	Expectation*: Worst Case:		
Insert	Expectation*: Worst Case:		
Remove	Expectation*: Worst Case:		
Storage Space			

Bonus Slides

Choosing a Hash Function

Python has a built-in hash! It's pretty good if you run everything at once.

```
1 print(hash("I can pass in any string!”))
2
3
4 print(hash(205811))
5
6
```

Choosing a Hash Function

If you want something that is persistently deterministic, find a seeded hash

```
1 import mmh3
2
3 print(mmh3.hash("I can pass in any string!", 10)) #I got: -565691678
4 print(mmh3.hash("I can pass in any string!", 50)) #I got: -947521776
5 print(mmh3.hash("I can pass in any string!", 12)) #I got: 1680496801
6
```

Hash Table

Worst-Case behavior is bad — but what about randomness?

1) **Fix *h***, our hash, and assume it is good for ***all keys***:

Simple Uniform Hashing Assumption (SUHA)

2) Create a ***universal hash function family:***

Hash Function (Division Method)

Hash of form: $h(k) = k \,\%\, m$

Pro:

Con:

Hash Function (Multiplication Method)

Hash of form: $h(k) = \lfloor m(kA \,\%\, 1) \rfloor,\ 0 \leq A \leq 1$

Pro:

Con:

Hash Function (Universal Hash Family)

Hash of form: $h_{ab}(k) = \big((ak + b)\,\%\,p\big)\,\%\,m,\ a, b \in Z_p^*, Z_p$

$$\forall k_1 \neq k_2,\ Pr_{a,b}(h_{ab}[k_1] = h_{ab}[k_2]) \leq \frac{1}{m}$$

Pro:

Con:

Where do we go from here?

Hash tables were a much needed detour (and allows for next week's lab to be a review session)

Still to discuss:

- Graph Algorithms
- Sets and hash table extensions
- 'Bonus topics'

Assignments remaining:

- This week's MP is last MP
- This week's lab is last lab

MP Algorithm

A trio of independent graph algorithm projects designed as a capstone

Learning Objectives:

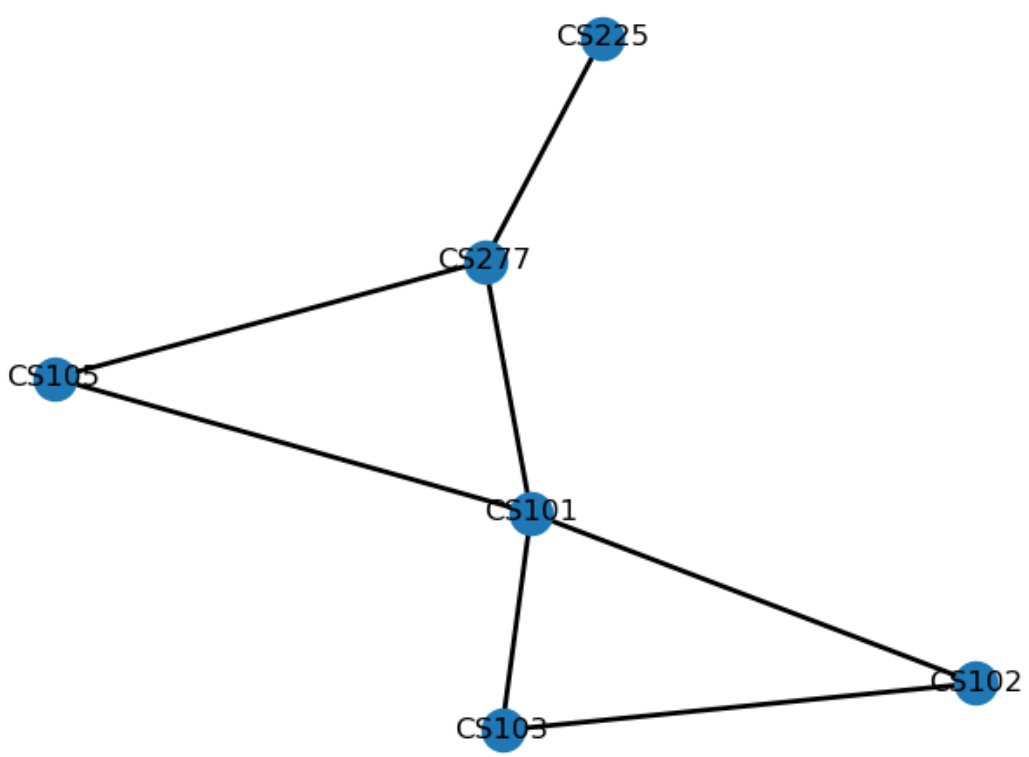
Practice parsing different data formats into graphs

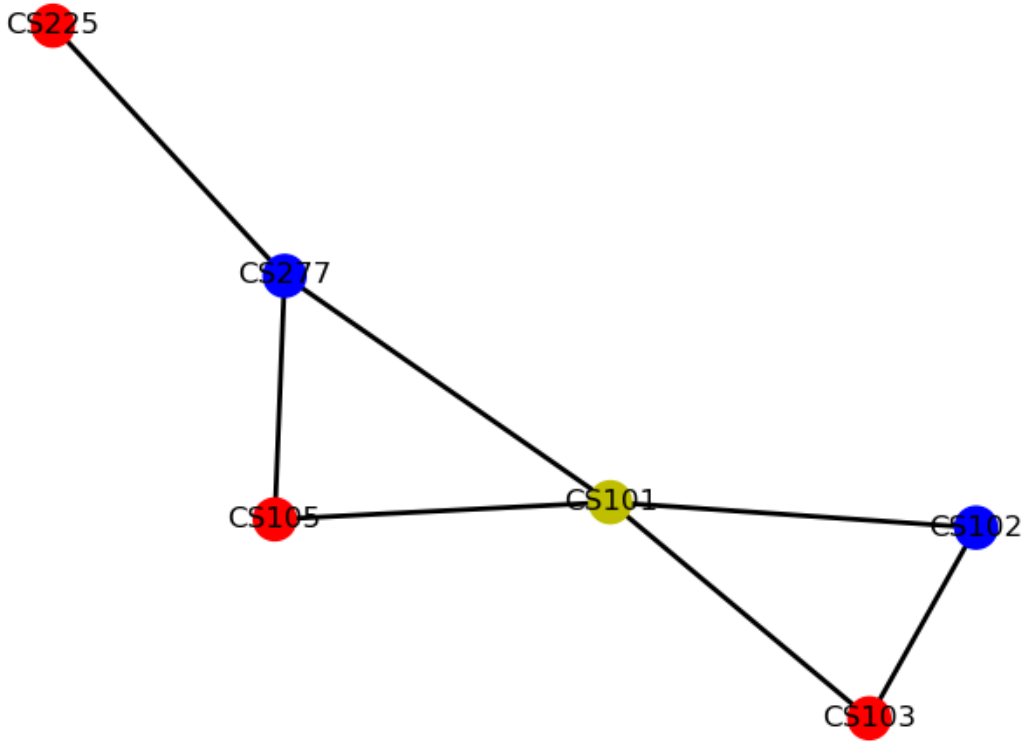
Practice fundamentals of accessing and modifying graphs in NetworkX

Create a greedy heuristic algorithm to solve a complex problem

Part 1: Graph Coloring

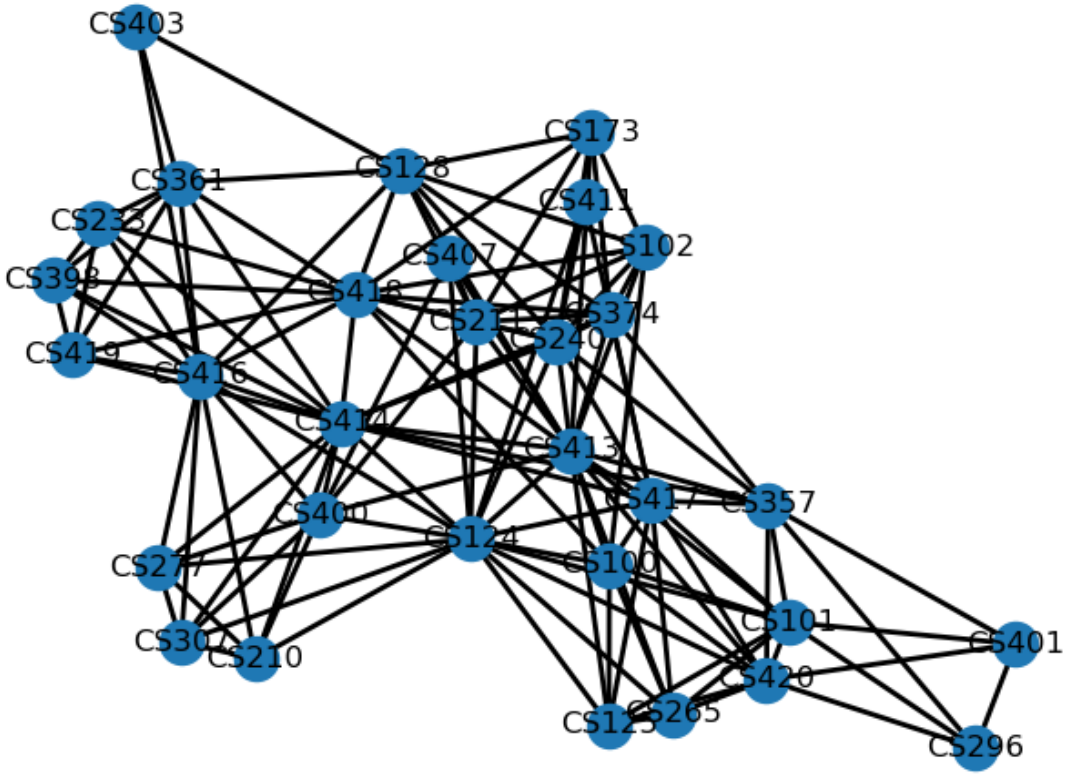
We want to assign a color label to every node in the graph such that no two neighbors have the same color

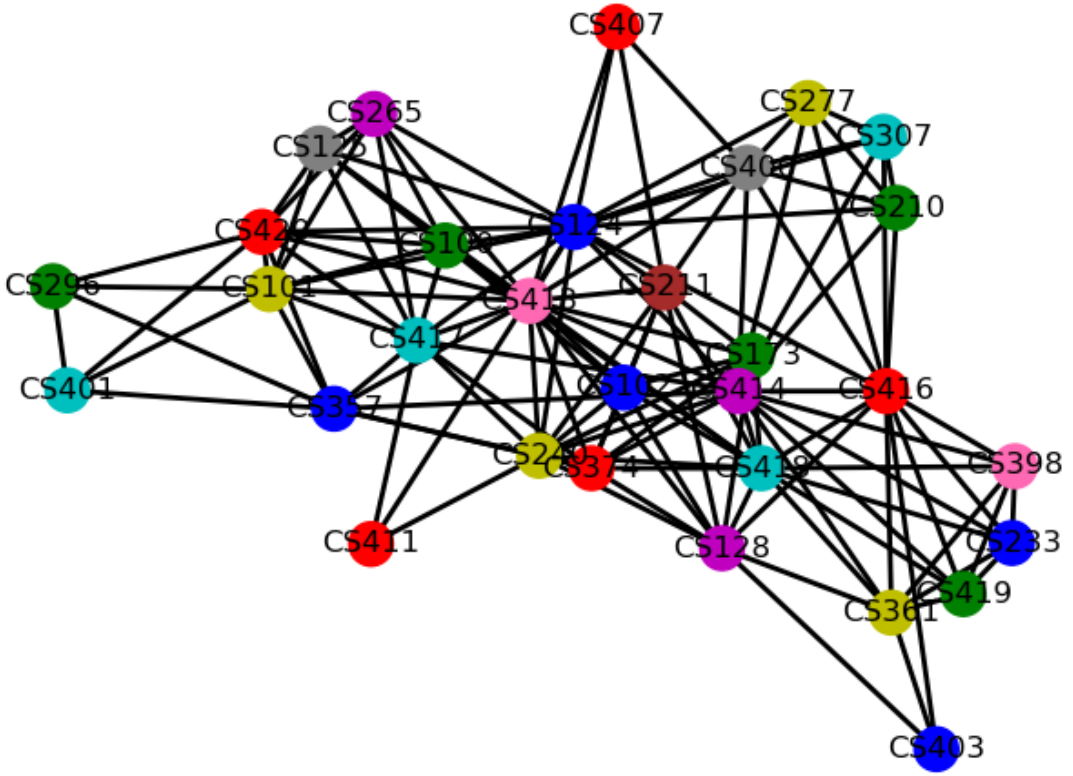




Part 1: Graph Coloring

If we want to minimize the number of colors, this can get very computationally intensive very quickly…





Part 1: Graph Coloring

We will do this using a greedy heuristic of our own design:

Given a graph, a list of vertices, and a list of colors

For each node in a specified order:

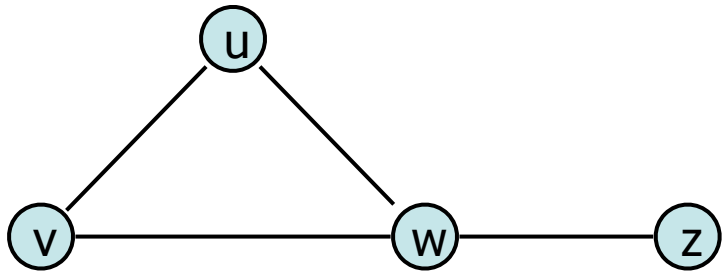
Check the color of every neighbor

Label the node the first unused color

Ex:

Nodes: V, U, W, Z

Color: R, G, B



Part 1: Graph Coloring

We will do this using a greedy heuristic of our own design:

Given a graph, a list of vertices, and a list of colors

For each node in a specified order:

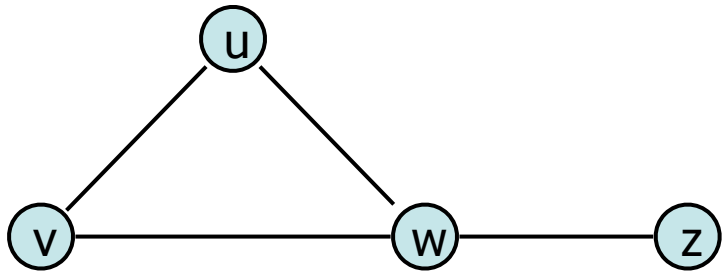
Check the color of every neighbor

Label the node the first unused color

Ex:

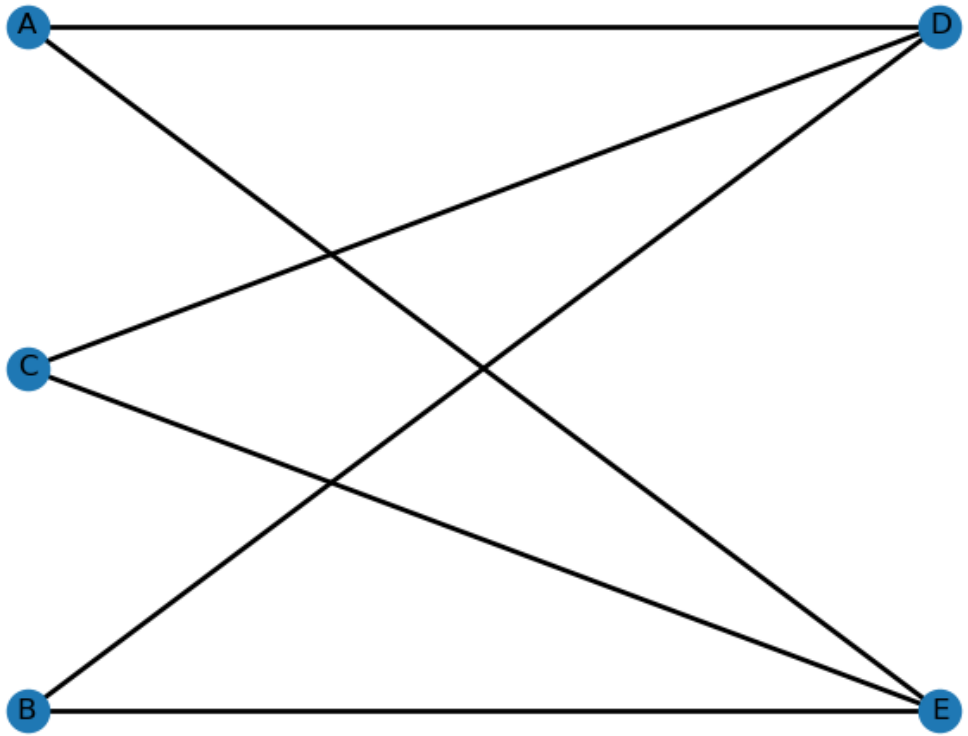
Nodes: W, V, U, Z

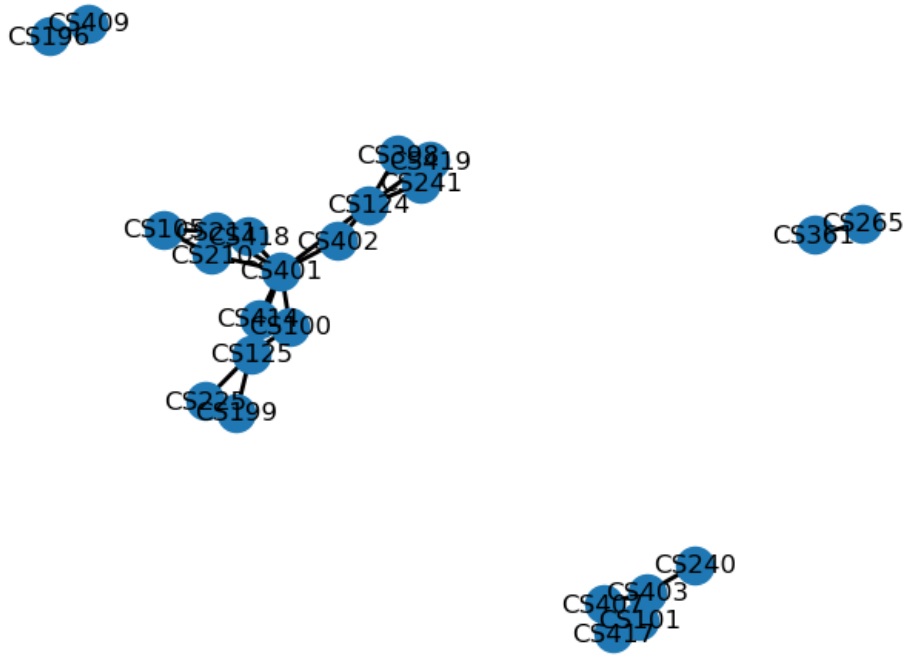
Color: R, G, B



Part 1: Graph Coloring

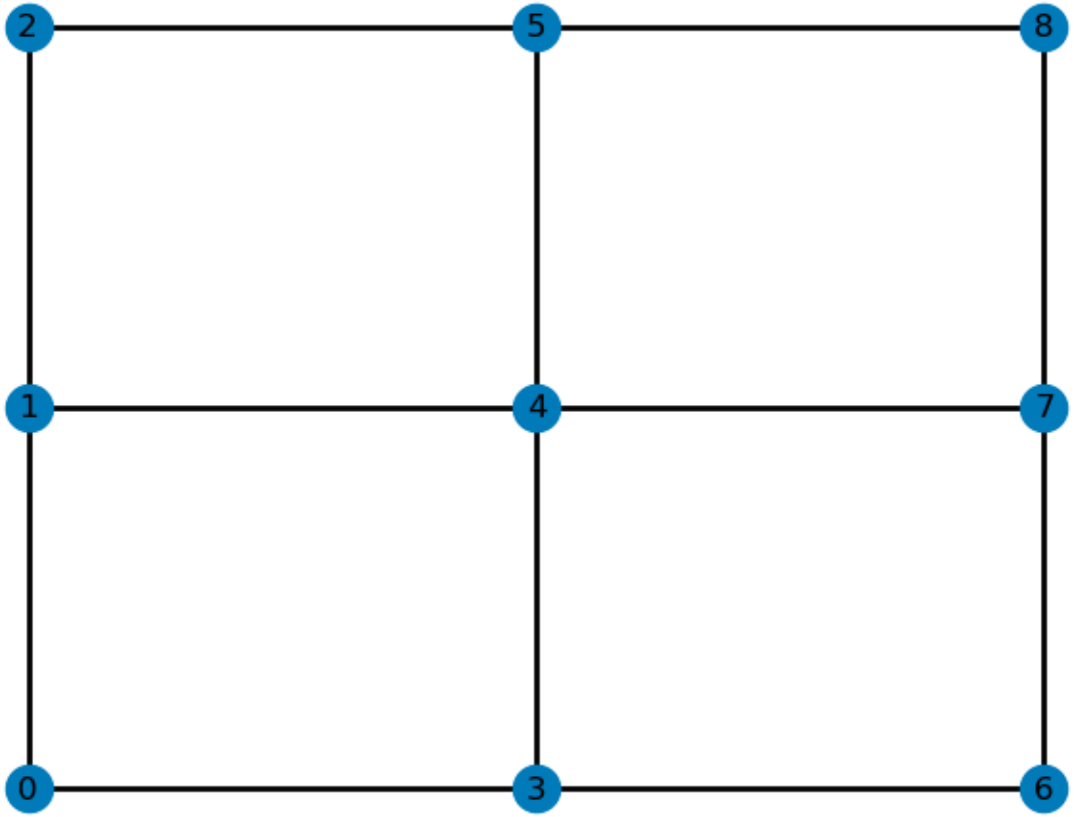
You are also responsible for making two different types of graphs:





Part 2: Pirate Walk (on a graph!)

Given a grid graph (with node attribute ‘pos’), a start node, and a path string, record the list of vertices the path goes through in order.



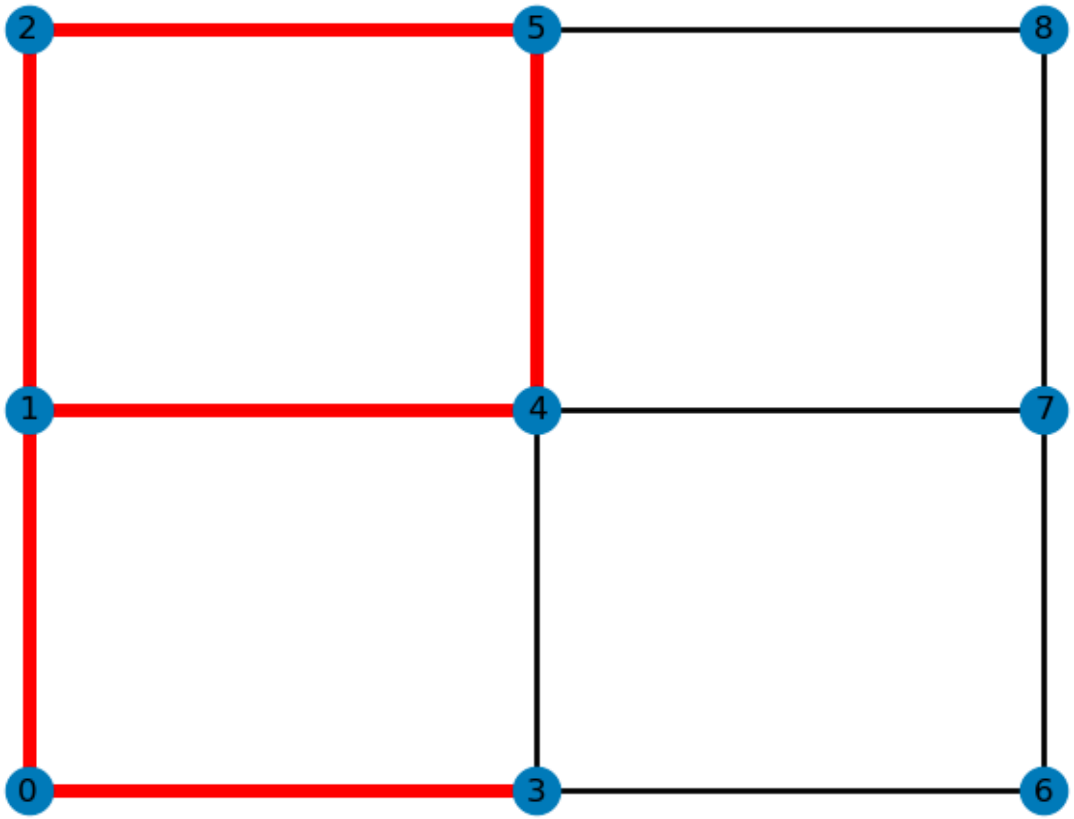
Start: 1

Path: “ENWSSE”

Output:

Part 2: Pirate Walk (on a graph!)

Given a grid graph (with node attribute ‘pos’), a start node, and a path string, record the list of vertices the path goes through in order.



Start: 1

Path: “ENWSSE”

Output: [1, 4, 5, 2, 1, 0, 3]

Part 3: OpenFlights Flight Paths

Given two csv files (vertex & edge), build a weighted NetworkX graph

645,"Haugesund Airport”,"Haugesund","Norway","HAU","ENHD",59.34529876709,5.2083601951599,86,1,"E","Europe/Oslo","airport","OurAirports"

11092,"Larned Pawnee County Airport","Larned","United States",\N,"KLQR",38.20859909,-99.08599854,2012,-5,"A",\N,"airport","OurAirports"

293,"Djerba Zarzis International Airport","Djerba","Tunisia","DJE","DTTJ",33.875,10.775500297546387,19,1,"E","Africa/Tunis","airport","OurAirports"

"KLQR","DTTJ"

“ENHD","KLQR"

“ENHD","DTTJ"

Solve each problem your own way

Part 3 (and to a lesser degree the overall assignment) has less structure than past assignments — by design!

Use what you've learned in the class previously to build graphs from different inputs

You can (and are encouraged to) freely discuss your approach to solving these problems