CS5002
3/13 - Week 10 !!

Admin

- no class next week! OH SH135, online
- HW6 out now, due on 3/27 6pm
- Exam #2 4/10, make-up Q 4/24

Agenda

1. Graph overview
2. Representation of graphs / Properties
3. LCA #7
4. Graph algorithms - searching
5. Graph algorithm - Dijkstra's (shortest path)

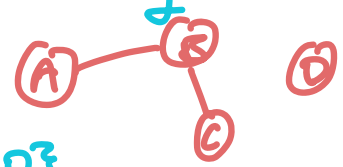
1. Graph Overview

A graph is a discrete structure
↳ collection of objects

Set — unordered collection of distinct objects {—, —, —, —}
— have things, count, probabilities

Graph — uses sets to represent more complexity {A, B, C, D}

↳ $G = (V, E)$



where V is a set of vertices {A, B, C, D}
E is a set of edges {(A, B), (B, C)}

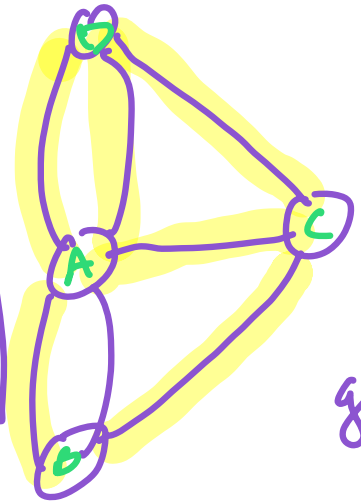
Graph origin story - Euler
Konigsberg

goal: visit every neighborhood
traverse each bridge
exactly once

map → graph

- every neighborhood = vertex
- every bridge = edge

You cannot visit every neighborhood crossing every bridge exactly once



vertex, cross an edge → next neighborhood

$V = \{A, B, C, D\}$

isn (B, C)

$E = \{(A, B), (A, C), (A, D), (C, D)\}$

get from A to B? How many bridges?
B to D?

- every map can be repped as a graph
- not every graph reps a map

} two types of problems

Today in CS:

- google maps, directions
- social network (relationships)
- recommendation systems
- dependency graph for OS/CPU

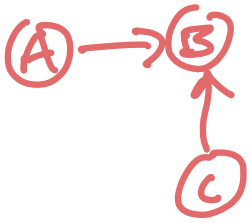
Definitions $G = (V, E)$

- edge (u, v) indicates that vertices u, v are adjacent

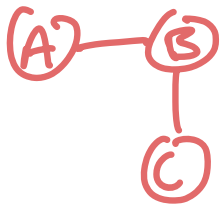
- V must be non-empty, E can be empty

valid graph

- A graph can be directed or undirected

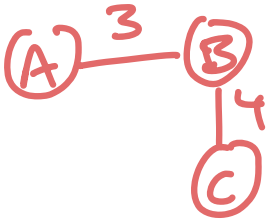


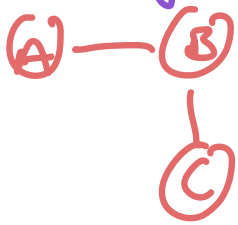
$E = \{(A,B), (C,B)\}$



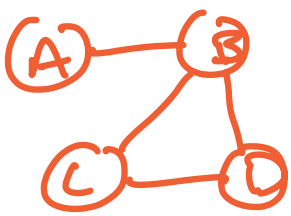
$E = \{(A,B), (B,C)\}$

- A graph can be <u>weighted</u> or <u>unweighted</u>





- A <u>path</u> is a sequence of vertices, where each successive pair is an edge

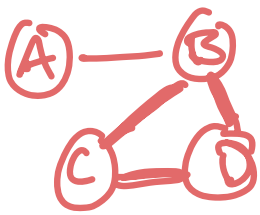


path from A to D:

A, B, D — shortest path

A, B, C, D

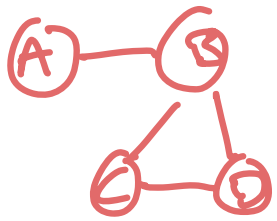
- A <u>cycle</u> is a non-empty path in which first == last



Cycle: B, C, D, B

(An acyclic graph contains no cycles)

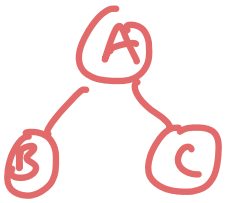
- A connected graph, there is a path between any two vertices



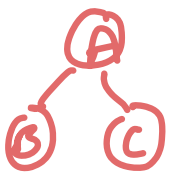
✓ connected!

not connected

- A tree is a connected graph with no cycles



- Degree of a vertex is the # of incident edges



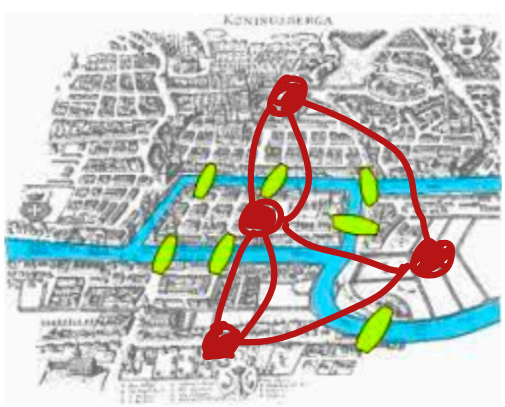
deg(A) = 2
deg(B) = 1
deg of graph = 4 (2x # edges)

- simple graph
  - unweighted
  - undirected
  - no multiedges
  - no self loops

7:00

2. Representation of Graphs
& and properties

- total deg = 2x # edges
  (14 total agree)
- every edge leaves one vertex,

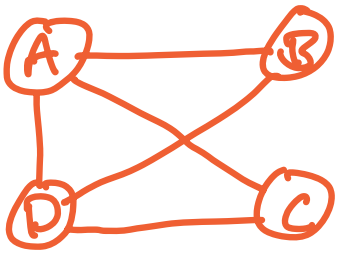


drives at another

- need to end at vertex with odd degree
- degree of each vertex affects paths you can take

degree 3: drive leave drive (any odd degree)

Simple graph

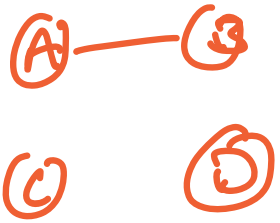


needs:
- adjacency list
- adjacency matrix

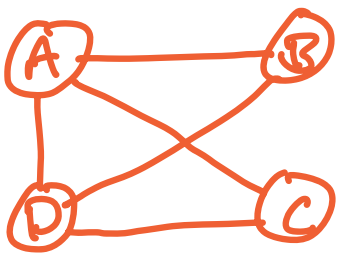
Adj list

- list every vertex
- every vertex contains its neighbors

A: B, C, D
B: A, D
C: A, D
D: A, B, C



A, B



(sum of row == degree of vertex)

Adj matrix

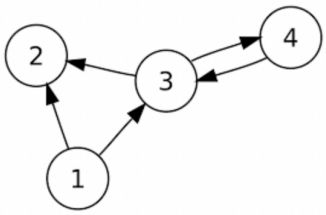
- table of all vertices
- indicate whether edge exists

0 = no edge
1 = edge

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

We use both reps
- matrix is good to look up edges, and for weighted graphs
- list is good for sparse graph

For this ICA, consider the graph below.



Problem #1

A. Is it directed or undirected?

directed

B. How many vertices does it have?

4

C. How many total edges does it have?

5

D. Which vertices have total degree 2?

1 2 4

→ directed graph
in-degree, out-degree

in-deg = 1
out-deg = 1
total = 2

E. What is a valid path from 1 to 4?

1, 3, 4

F. Is the graph connected or unconnected? (Fun fact: in this kind of graph we would say "strongly connected" rather than connected, but it has the same definition)

not connected!

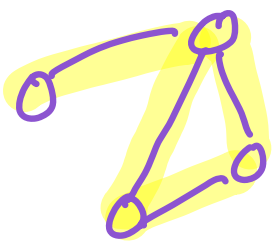
9:45

Problem #2

How would you represent the graph in an adjacency list?

1: 2 3
2:
3: 2 4
4: 3

total deg = 10



eulerian path:
2 vertices of odd degree
(or none)

3. Graph Algorithms - Searching

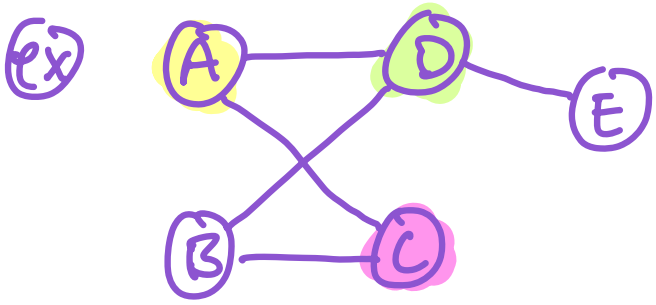
Searching a graph:

- start at some vertex, how do I get to another one?
- start at some vertex, where can I get from here?

(1) Breadth-First Search (BFS)

- start at some vertex
- find all vertices reachable in one step (neighbors)
  - find all vertices reachable in 2 steps (neighbors' neighbors)

...



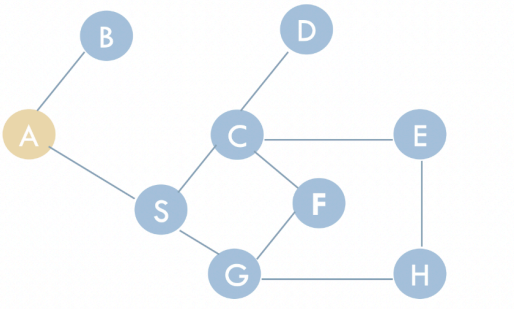
A: C, D
B: C, D
C: A, B
D: A, B, E
E: D

A, C, D, B, E
1 step 2 steps

BFS, start at A

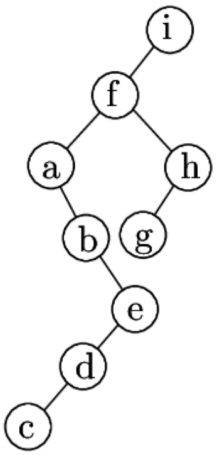
- write down order in which we visit vertices
- break ties alphabetically

BFS start at A



A, B, S, C, G, D, E, F, H

BFS start at i



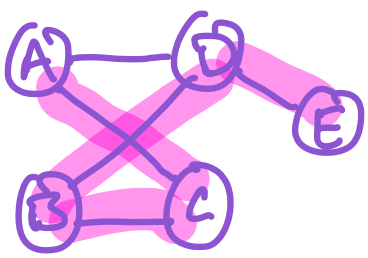
i, f, a, h, b, g, e, d, c

1 skip 2 3

BFS gives shortest path from starting vertex to every reachable vertex in a simple graph

2. Search - Depth First Search (DFS)

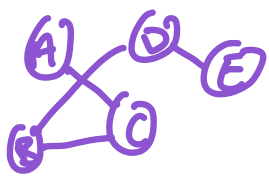
- start at given vertex
- go down a path as far as possible, until a dead end
- at dead end, back track as little as possible



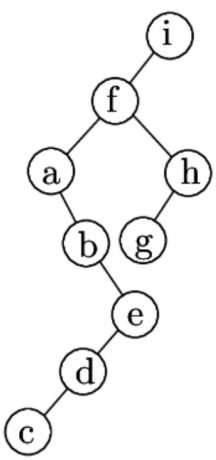
DFS, start at A (alphabetical)

A, C, B, D, E

result:

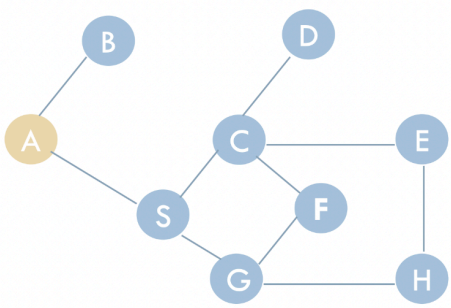


DFS start at i



i, f, a, b, e, d, c, h, g

DFS start at A



A, B, S, C, D, E, H, G, F

DFS result:

- DFS tree (get rid of extra edges)

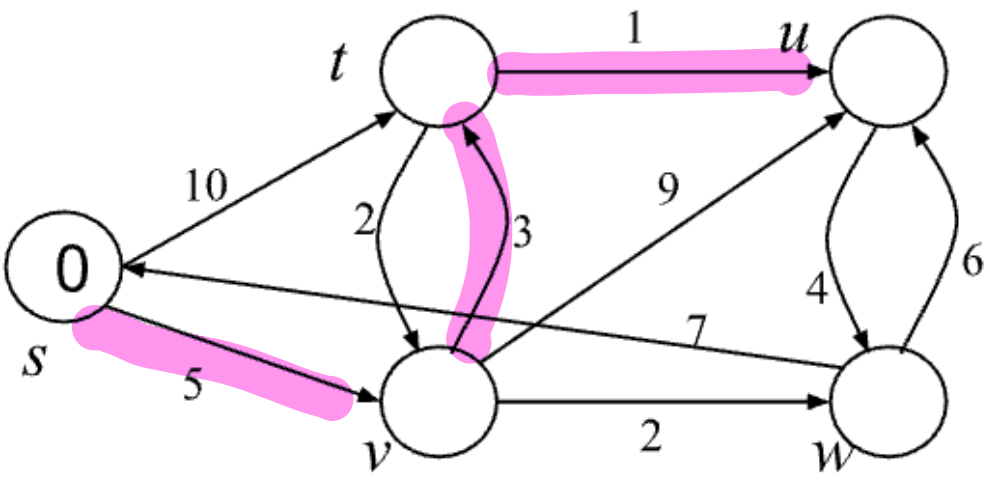
8:35

4. Dijkstra's Algorithm

- BFS gives shortest path on simple graph
- Dijkstra's gives shortest path on directed, weighted graph
  - ↳ non-negative

Start at given vertex
- want to get to every reachable vertex
- every path has total weight, want smallest
- visit the neighbors, assume shortest path
  - ↳ then, try for better way
  - "relax" edges
  - start with smallest total path and see if the overall path gets better

As we go, track:
- predecessor in path from start
- current weight of path from start
- process "next" vertex, one with shortest path so far



Dijkstra's, start at s

	s	t	u	v	w
pred	/	~~s~~ v	~~v~~ ~~t~~	s	v
weight	0	~~10~~ 8	~~14~~ ~~13~~ 9	5	7

#1 start with s

#2 next shortest is v

#3 next shortest is w

#4 next shortest is t

#5 next shortest is u
no updates

↓

	s	t	u	v	w
pred	/	v	t	s	v
weight	0	8	9	5	7