CS1800 Day 15

Admin:
- HW5 (probability) due today
- HW6 (graphs) released today
- "Extra" video on BFS / DFS (see website)
- might end few mins early today, feel free to hang out if you have BFS / DFS or Dijkstra questions

`

Content:

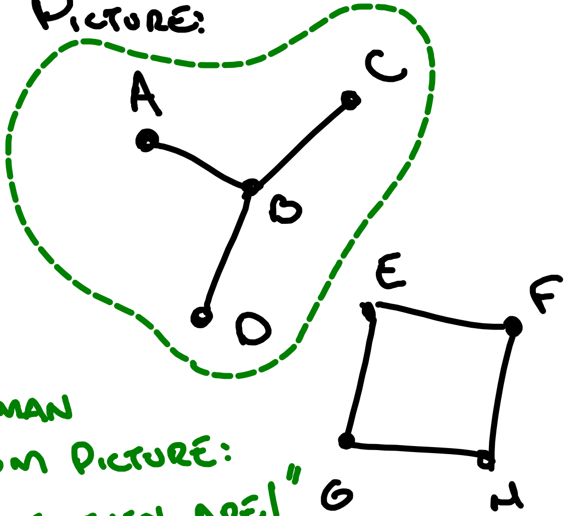
Searching through all the nodes in a graph:
- Breadth First Search (BFS)
- Depth First Search (DFS)

Finding the shortest path between two nodes in a weighted graph:
- Dijkstra's Algorithm

Searching a graph: (BFS & DFS intro)

Goal: Using a computer, walk (order) to all nodes which are connected to node A

Picture:



Human from picture:
"Here they are!"

Neighbor lists

A: [B]
B: [A C D]
C: [B]
D: [B]
E: [F G]

Computer from representation:
"Not so simple..."

F: [E H]
G: [E H]
H: [G F]

Depth First Search: Inuition & Animation

Approach: "visit an adjacent, unvisited node as long as possible,
then backup one edge and look for another vertex to visit, using a depth first search."

<view gif>

gif source: https://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html

Breadth First Search: Intuition & Animation

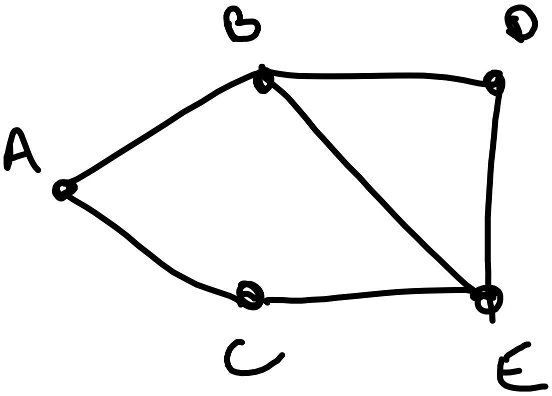
Approach: "Visit all the vertices adjacent to the starting vertex,
then do a breadth first search from each of those vertices."

<view gif>

gif source: https://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html

Breadth First Search: Example

Approach: "Visit all the vertices adjacent to the starting vertex,
then do a breadth first search from each of those vertices."



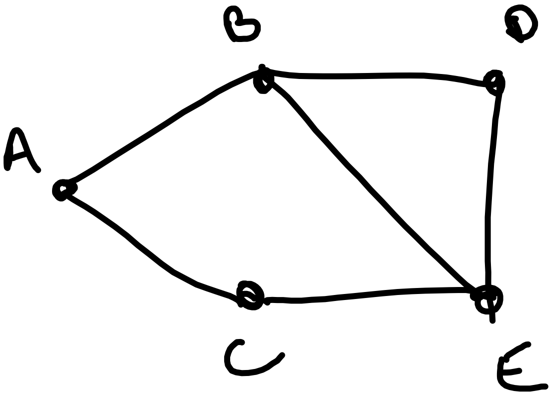
VISITED:

A

BFS / DFS require some starting node be given, where the search is initialized.

Breadth First Search: Example

Approach: "Visit all the vertices adjacent to the starting vertex,
then do a breadth first search from each of those vertices."



VISITED:

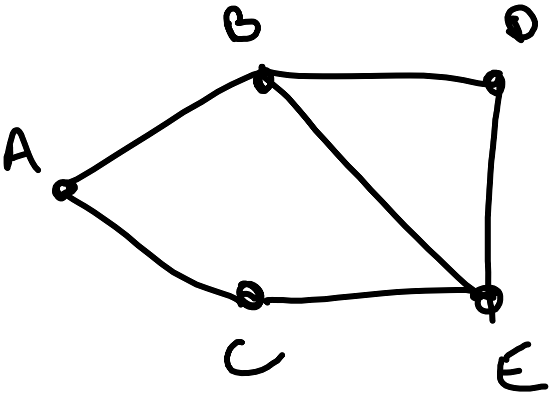
A BC

CURRENT NODE

A's neighbors are {B, C}. We could visit them in any order (BC or CB) as a valid BFS. We choose alphabetical ordering to standardize output

Breadth First Search: Example

Approach: "Visit all the vertices adjacent to the starting vertex,
then do a breadth first search from each of those vertices."



VISITED:

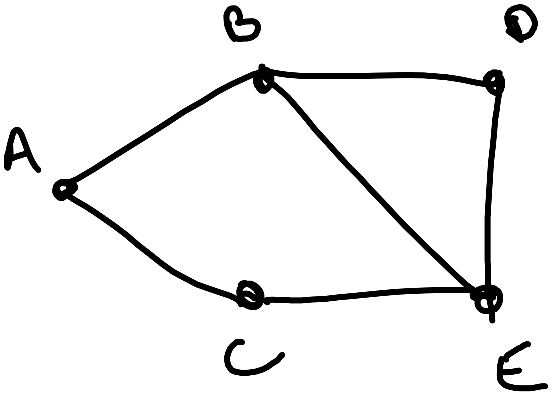
A B C D E

CURRENT NODE

B's neighbors are {A, D, E} but we only add the unvisited nodes to our list (again in alpha order)

Breadth First Search: Example

Approach: "Visit all the vertices adjacent to the starting vertex,
then do a breadth first search from each of those vertices."



VISITED:

A B C D E

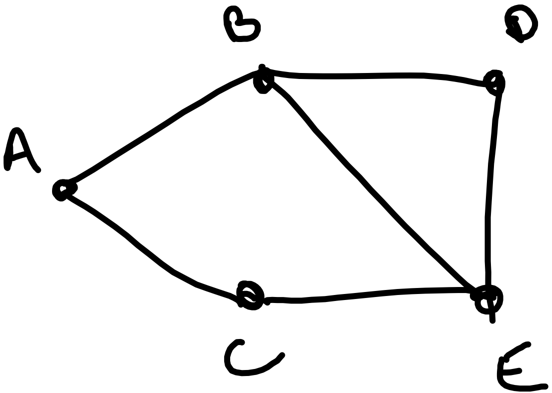
CURRENT NODE

(C has no unvisited neighbors to add)

Looking at the picture, you can tell we're done.
The computer doesn't know ... must finish BFS on visited list

Breadth First Search: Example

Approach: "Visit all the vertices adjacent to the starting vertex,
then do a breadth first search from each of those vertices."



VISITED:

A B C D E

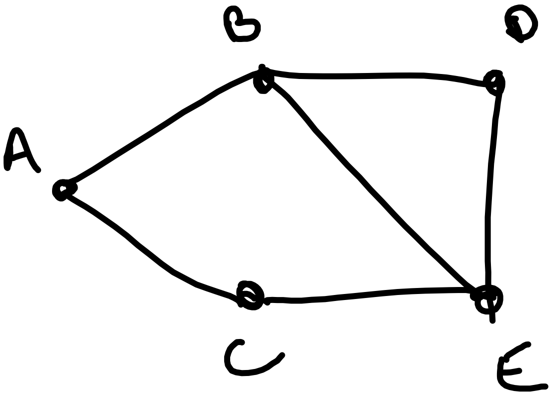
CURRENT NODE

(D has no unvisited neighbors to add)

Looking at the picture, you can tell we're done.
The computer doesn't know ... must finish BFS on visited list

Breadth First Search: Example

Approach: "Visit all the vertices adjacent to the starting vertex,
then do a breadth first search from each of those vertices."



VISITED:

A B C D E

(E has no unvisited neighbors to add)

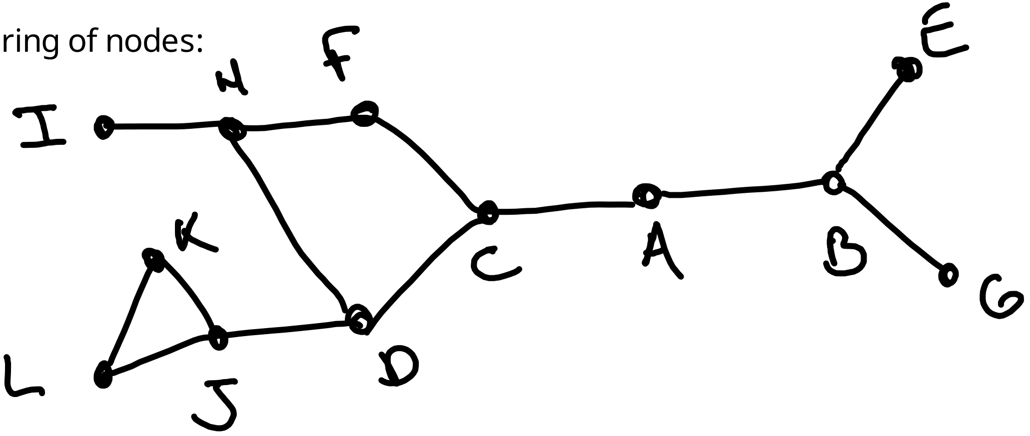
CURRENT NODE

Looking at the picture, you can tell we're done.
The computer doesn't know ... must finish BFS on visited list

In Class Activity: Breadth First Search

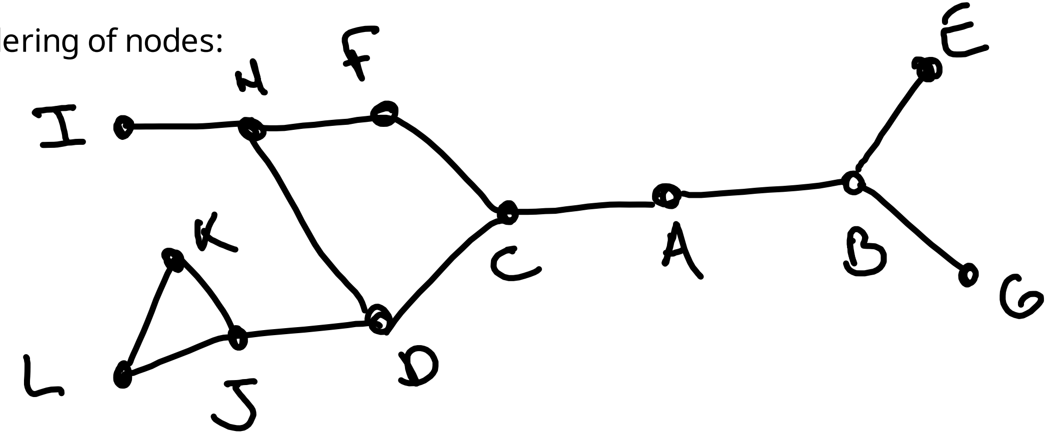
Give the BFS ordering of nodes:

- starting at A
- starting at H
- starting at G



In Class Activity: Breadth First Search

Give the BFS ordering of nodes:
- starting at A
- starting at H
- starting at G

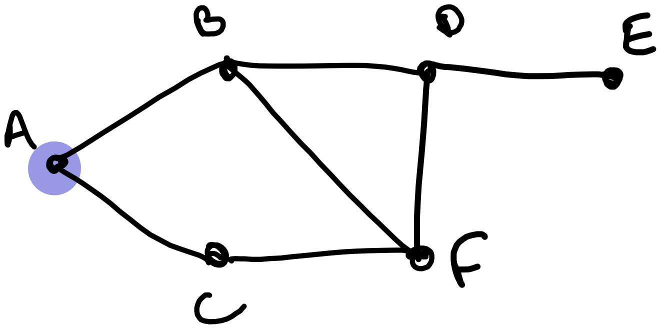



```
BFS start @ a: ABCE GDFH JIKL
BFS start @ h: HDFI CJAK LBEG
BFS start @ g: GBAE CDFH JIKL
```

Depth First Search: Example

Approach: "visit an adjacent, unvisited node as long as possible,
then backup one edge and look for another vertex to visit, using a depth first search."

PATH

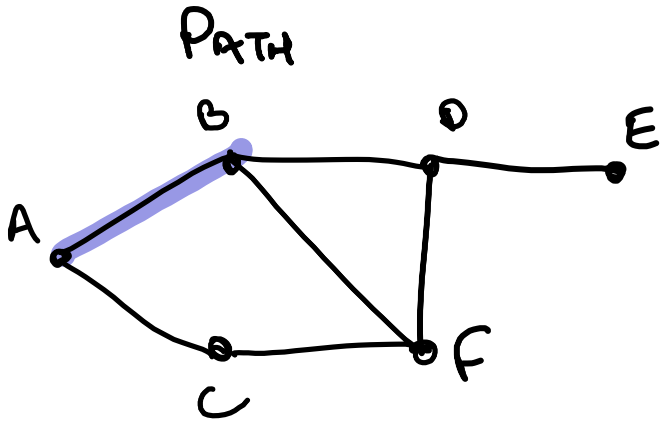


VISITED:

A

Depth First Search: Example

Approach: "visit an adjacent, unvisited node as long as possible,
then backup one edge and look for another vertex to visit, using a depth first search."



VISITED:

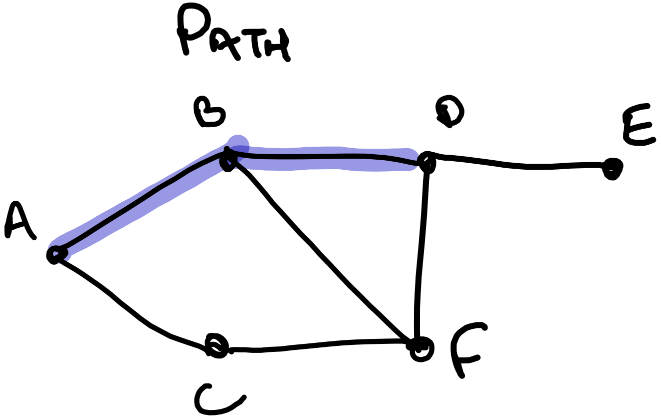
A B

A has two unvisited neighbors {B, C}

Again, we choose to visit the one which is alphabetically first

Depth First Search: Example

Approach: "visit an adjacent, unvisited node as long as possible,
then backup one edge and look for another vertex to visit, using a depth first search."



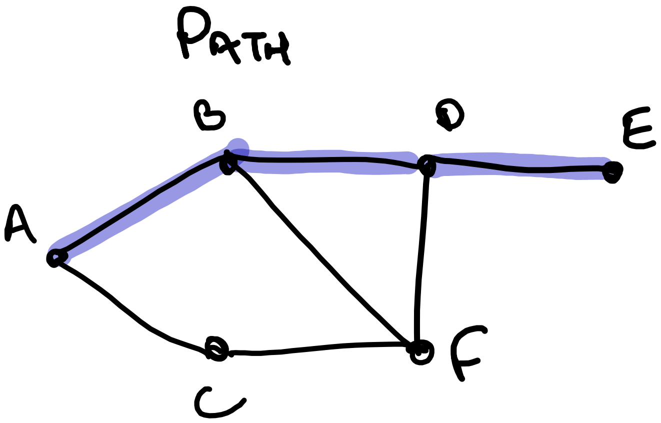
VISITED:

A B D

B has two unvisited neighbors {D, F},
we choose the one which is alphabetically first.

Depth First Search: Example

Approach: "visit an adjacent, unvisited node as long as possible,
then backup one edge and look for another vertex to visit, using a depth first search."



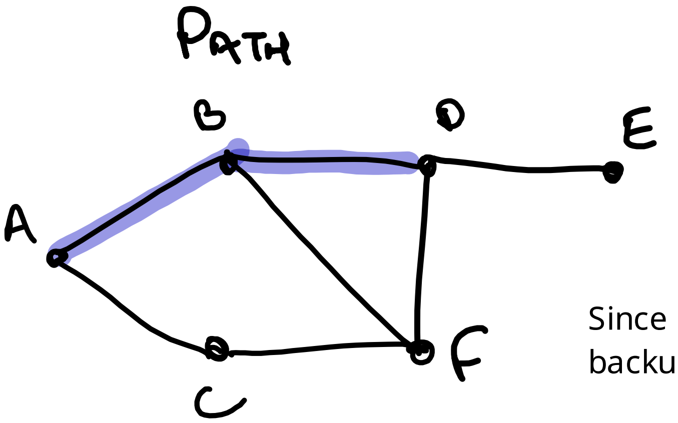
VISITED:

A B D E

D has two unvisited neighbors {E, F},
we choose the one which is alphabetically first.

Depth First Search: Example

Approach: "visit an adjacent, unvisited node as long as possible,
then backup one edge and look for another vertex to visit, using a depth first search."



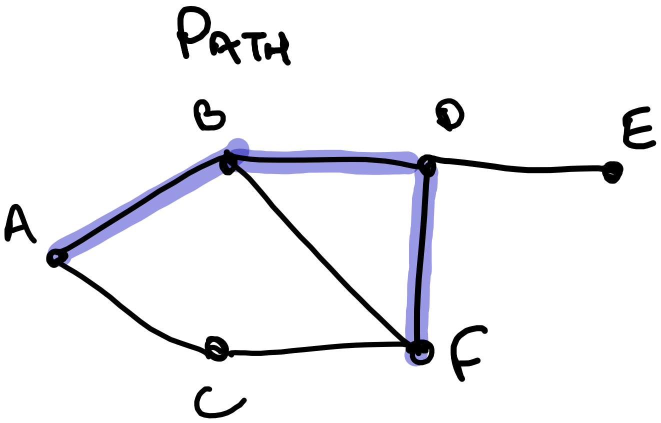
VISITED:

A B D E

Since E has no unvisited neighbors, we backup our path and repeat the DFS process

Depth First Search: Example

Approach: "visit an adjacent, unvisited node as long as possible,
then backup one edge and look for another vertex to visit, using a depth first search."



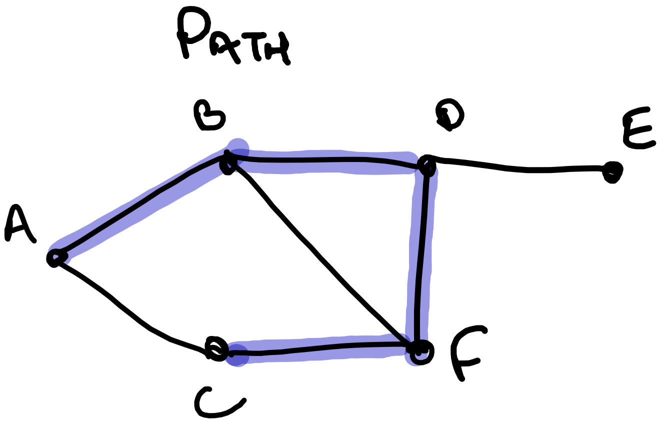
VISITED:

A B D E F

D has 1 unvisited neighbor {F}

Depth First Search: Example

Approach: "visit an adjacent, unvisited node as long as possible,
then backup one edge and look for another vertex to visit, using a depth first search."



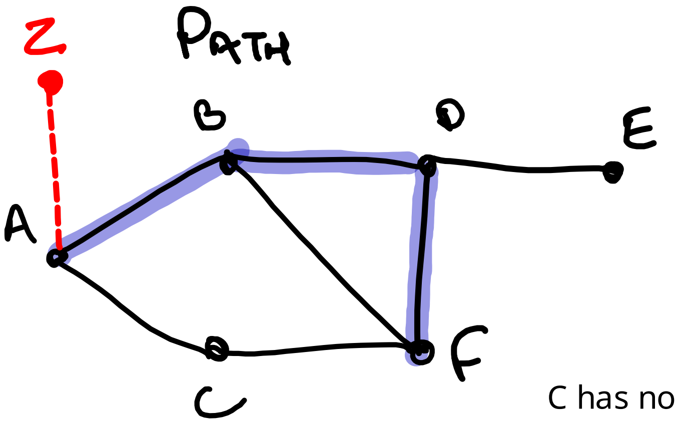
VISITED:

A B D E F C

F has 1 unvisited neighbor {C}

Depth First Search: Example

Approach: "visit an adjacent, unvisited node as long as possible,
then backup one edge and look for another vertex to visit, using a depth first search."



VISITED:

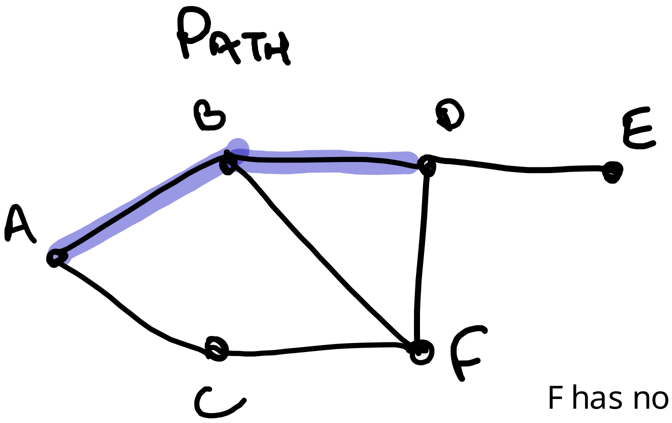
A B D E F C

C has no unvisited neighbors so we backup

(You can tell from the picture we're done, the computer can't ...
... we would've arrived at this step if a "z-node" had been present all along)

Depth First Search: Example

Approach: "visit an adjacent, unvisited node as long as possible, then backup one edge and look for another vertex to visit, using a depth first search."



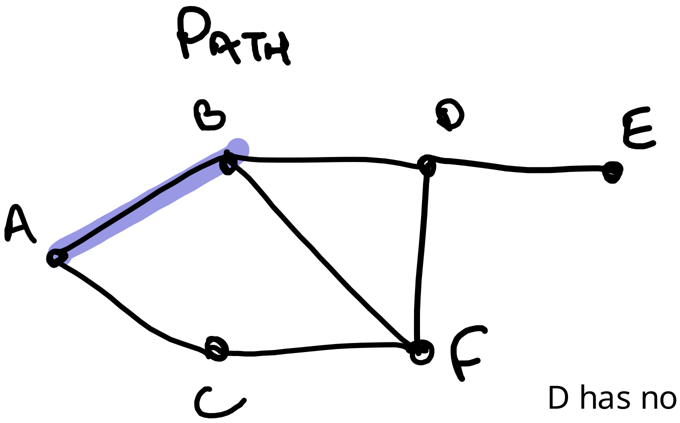
VISITED:

A B D E F C

F has no unvisited neighbors so we backup

Depth First Search: Example

Approach: "visit an adjacent, unvisited node as long as possible,
then backup one edge and look for another vertex to visit, using a depth first search."



VISITED:

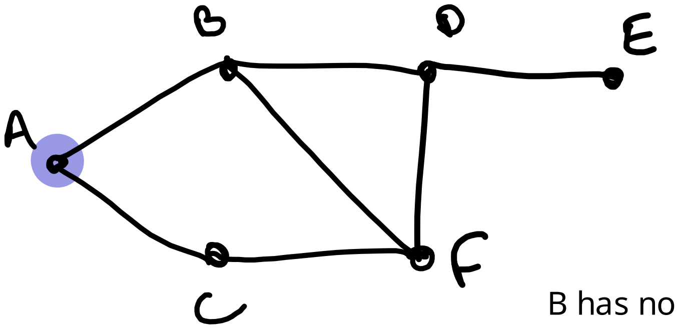
A B D E F C

D has no unvisited neighbors so we backup

Depth First Search: Example

Approach: "visit an adjacent, unvisited node as long as possible,
then backup one edge and look for another vertex to visit, using a depth first search."

PATH



VISITED:

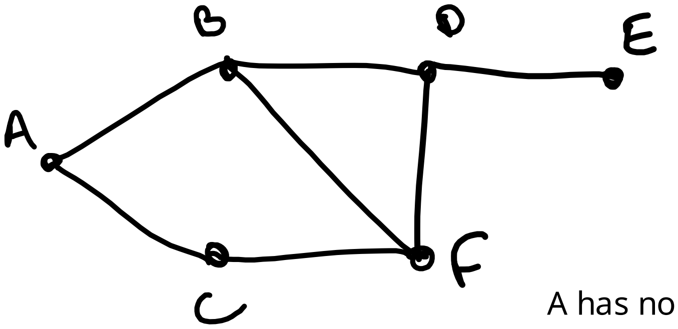
A B D E F C

B has no unvisited neighbors so we backup

Depth First Search: Example

Approach: "visit an adjacent, unvisited node as long as possible,
then backup one edge and look for another vertex to visit, using a depth first search."

PATH



VISITED:

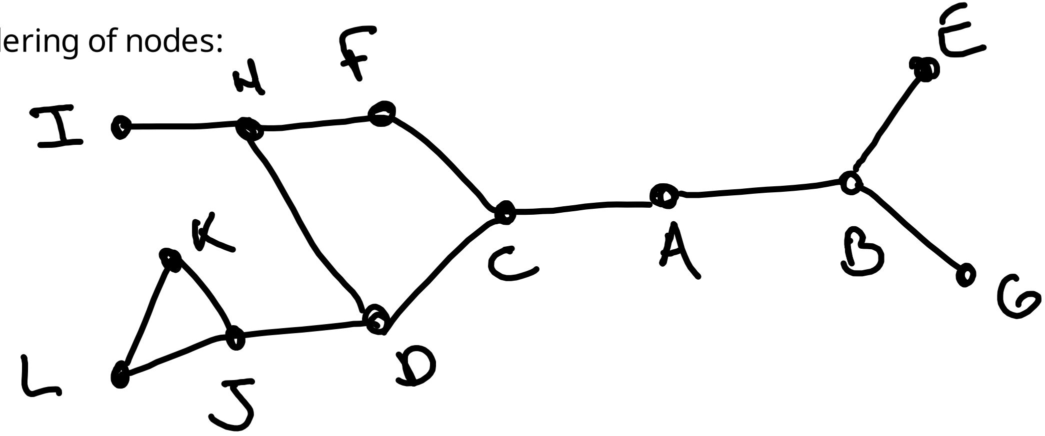
A B D E F C

A has no unvisited neighbors so we backup ...
... but we can't backup as A was our starting node.
DFS is complete

In Class Activity: Depth First Search

Give the DFS ordering of nodes:

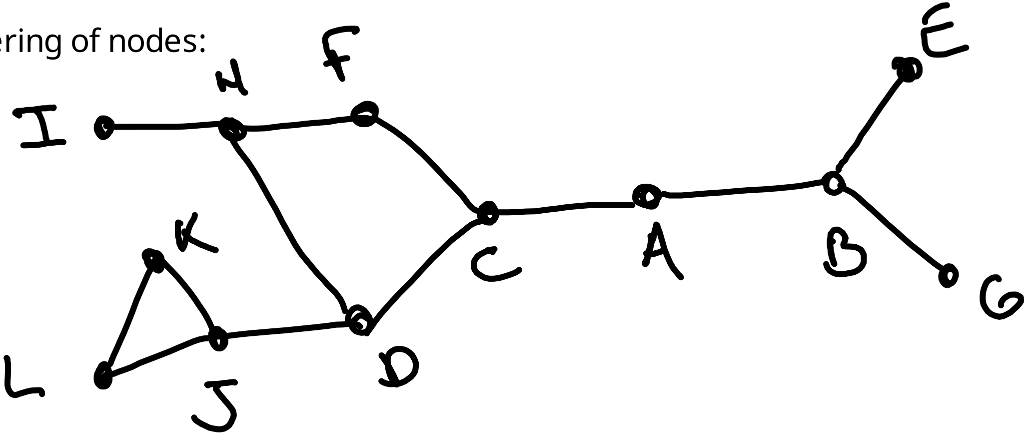
- starting at A
- starting at H
- starting at G



In Class Activity: Depth First Search

Give the DFS ordering of nodes:

- starting at A
- starting at H
- starting at G



DFS start @ A: ABEG CDHF IJKL

DFS start @ H: HDCA BEGF JKLI

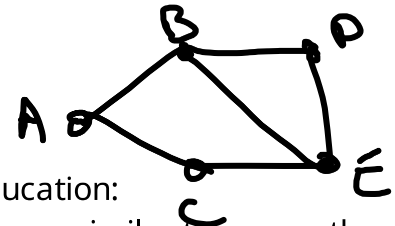
DFS start @ G: GBAC DHFI JKLE

BFS / DFS: Why did we do this again?

- BFS/DFS gives you the largest, connected subgraph
  - "What are all the cities I can get to taking flights from only one airline?"
  - computer can tell if a graph is connected
  - one run gives one connected component ... repeat again from univisited node for others

- DFS detects cycles in a graph
  - cycle exists if and only if we bump into a neighbor which has already been visited

- BFS orders all nodes from nearest to furthest starting point



BFS ORDERING: A B C D E
PATH LENGTH FROM A: 0 1 1 2 2

- Comp Sci Education:
  - They're very similar to many other graph algorithms
  - They can be built recursively (a function which calls itself). super useful pattern

Reminder:

Take a peek at the BFS / DFS extra video (next to today's notes on webpage)

In 10 minutes you will:

- see a more formulaic approach to BFS / DFS
    - useful if you, like me, forget what has / hasn't been visited

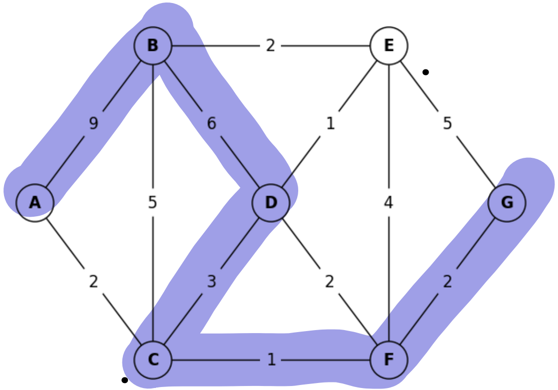
- be introduced to queues / stacks
- see how a computer organizes information as it runs BFS / DFS

SUPER USEFUL DOWN THE ROAD!

Shortest Path Problem

What path (sequence of unique, adjacent edges) has the lowest total cost from A to G?

Motivation: Suppose each node is a location and the edges weights are times to travel between the location. The shortest path gets us from A to G in the least time



An example path from A to G (not shortest):

A → B → D → C → F → G

$9 + 6 + 3 + 1 + 2 = 21$

TOTAL PATH COST (WEIGHT)

Shortest Path Problem

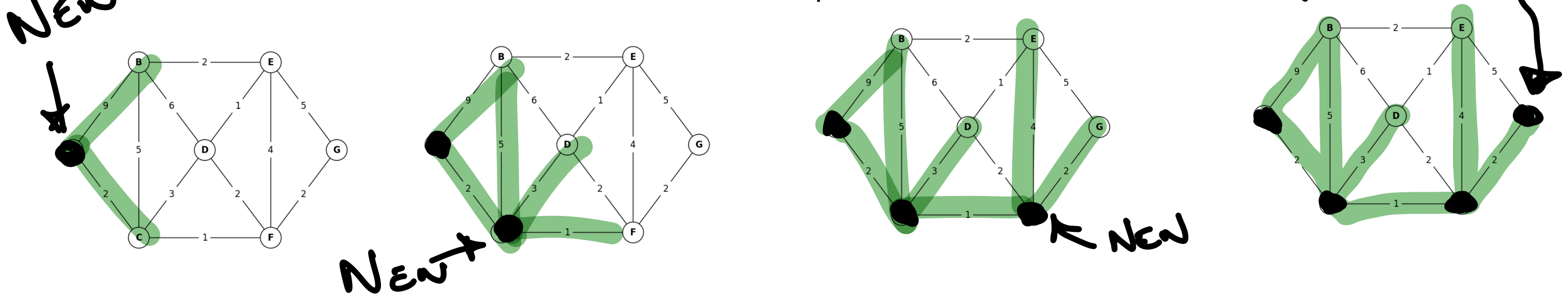
What path (sequence of unique, adjacent edges) has the lowest total cost from A to G?

Approach:

- Maintain a list of minimum-path-cost to a subgraph of nodes
- At every step, add new node (and its edges) to subgraph, choose node with current minimum-path-cost

Why it works:

- the minimum-path-cost of an added node is minimized over all paths in graph
    - (if there were another path with smaller cost, we'd be adding this one instead)
- when our destination node would be added, the path cost to it must be minimized



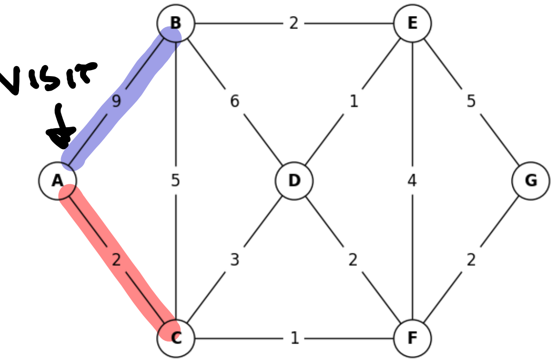
Shortest Path From A to G

Approach:
Update a table of min-cost-to-node for every node

visit node A:
Examine all edges to unvisited nodes:
- new destination? add cost to table

	A	B	C	D	E	F	G
Visited?	X						
Cost	0	9	2				

We always visit starting node first



The 9 in this table means there is a path from our starting node (A) to node B with a cost of 9.

Note: the 9 does not specify what this path is (more on this later)

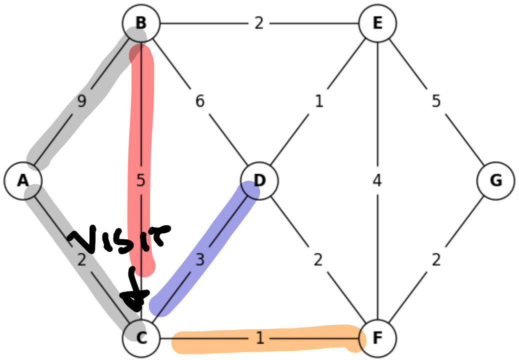
Shortest Path From A to G

Approach:
Update a table of min-cost-to-node for every node

visit node C:
Examine all edges to unvisited nodes:
- new destination? add cost to table

	A	B	C	D	E	F	G
Visited?	X						
Cost	0	9	2	5		3	

next node to visit: unvisited node with minimum cost
(C has cost 2, B has cost 9)



D is a new destination, add its cost to the table:
- A to C has cost 2 (from table above)
- C to D has cost 3 (from graph)
- A to D (through C) has cost 2 + 3 = 5

F is a new destination, add its cost to the table:
- A to C has cost 2 (from table above)
- C to F has cost 1 (from graph)
- A to F (through C) has cost 2 + 1 = 3

Shortest Path From A to G

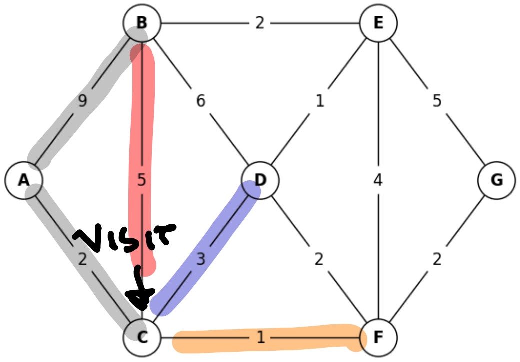
Approach:
Update a table of min-cost-to-node for every node

visit node C:
Examine all edges to unvisited nodes:

- new destination? add cost to table
- old destination w/ lower cost? update cost in table
- old destination w/ higher/equal cost? ignore this path

	A	B	C	D	E	F	G
Visited?	x		x				
Cost	0	7	2	5		3	

... we're still visiting C on this slide



Our new path to B:

- A to C has cost 2 (from table)
- C to B has cost 5 (from graph)
- A to B (through C) has cost 2 + 5 = 7

Our old path to B (read directly from table):

- some path exists to B with cost 9

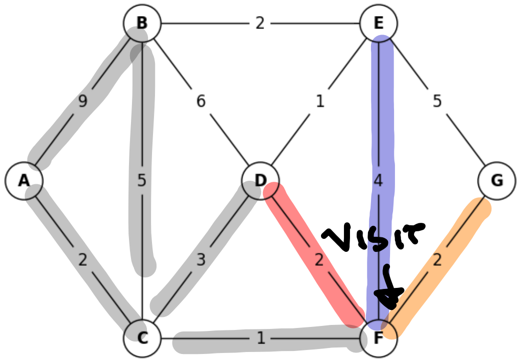
Shortest Path From A to G

Approach:
Update a table of min-cost-to-node for every node

visit node F:
Examine all edges to unvisited nodes:
- new destination? add cost to table
- old destination w/ lower cost? update cost in table
- old destination w/ higher/equal cost? ignore this path

	A	B	C	D	E	F	G
Visited?	x		x			x	
Cost	0	7	2	5	7	3	5

next node to visit: unvisited node with minimum cost
(B has cost 7, D has cost 5, F has cost 3)



E is a new destination: 3 to get to F (table) + 4 (F to E) = 7

G is a new destination: 3 to get to F (table) + 2 (F to G) = 5

old path to D: 5 (table)
new path to D: 3 to get to F (table) + 2 (F to D) = 5
we ignore this new path, it doesn't get added to table

Shortest Path From A to G

Approach:
Update a table of min-cost-to-node for every node

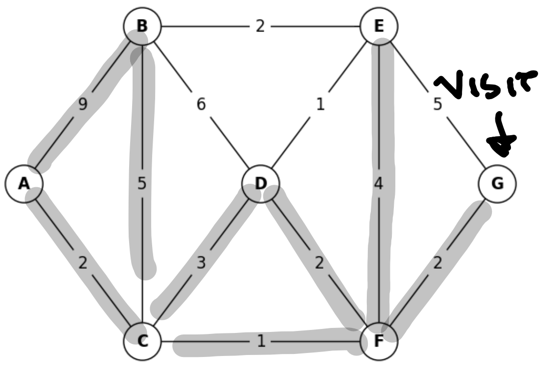
"visit" node G:

since our next node to visit has minimum cost
we stop the algorithm, we have our shortest path!

	A	B	C	D	E	F	G
Visited?	x		x			x	
Cost	0	7	2	5	7	3	5

next node to visit: unvisited node with minimum cost

(B has cost 7, D has cost 5, E has cost 7, G has cost 5)



Stop Algorithm:
Node G, our destination, has minimum cost among unvisited node:

there exists a path from A to G with cost 5

claim: this cost of 5 is minimum (no path with smaller cost from A to G exists in graph). See next slide for justification

Shortest Path Problem

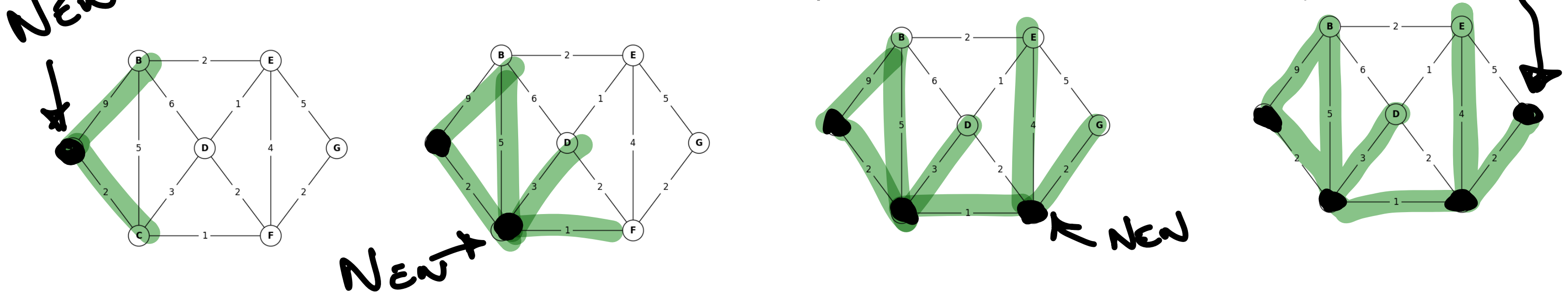
What path (sequence of unique, adjacent edges) has the lowest total cost from A to G?

Approach:

- Maintain a list of minimum-path-cost to a subgraph of nodes
- At every step, add new node (and its edges) to subgraph, choose node with current minimum-path-cost

Why it works:

- the minimum-path-cost of any newly visited node is minimized over all paths in graph
    - (if there were another path with smaller cost, we'd be visiting it instead)
- when our destination node would be added, the path cost to it must be minimized

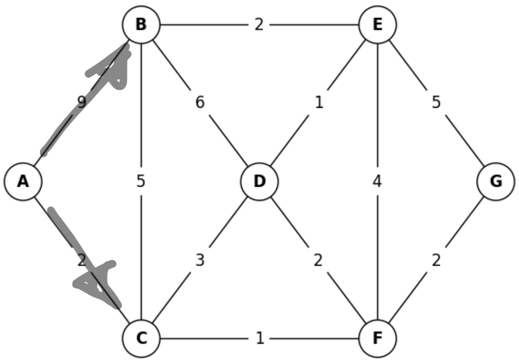


Wait ... the minimum cost form A to G is 5 but whats the path?

Lets go back and track each node's predecessor

(the node immediatley before itself on the shortest path from the starting node)

Shortest Path From A to G



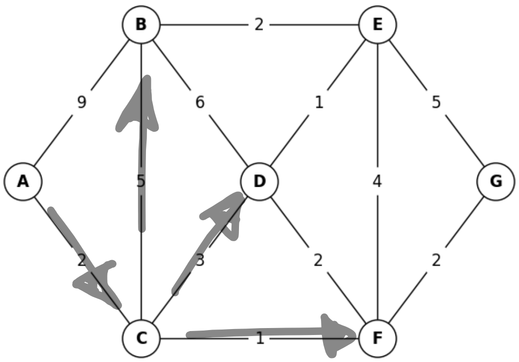
VISITING A

	A	B	C	D	E	F	G
Visited?	X						
Cost	0	A: 9	A: 2				

B's predecessor is A.
That is, this cost of 9 is achieved by:

- some path from our starting node to predecessor
- the edge from the predecessor to this node (A -> B)

Shortest Path From A to G



VISITING C

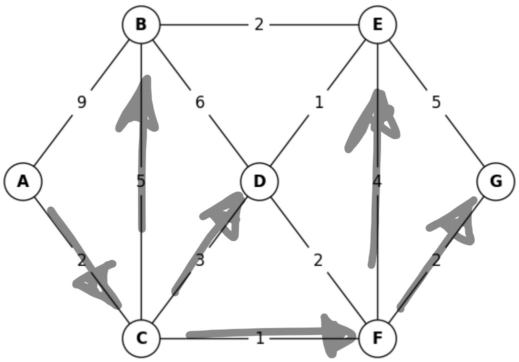
	A	B	C	D	E	F	G
Visited?	X		X				
Cost	0	C:7	A:2	C:5		C:3	

Notice: B's new predecessor is C.
That is, this cost of 7 is achieved by:

- some path from our starting node to predecessor
- the edge from the predecessor to this node (B -> C)

By recording the predecessor we record that path (A, C, B) has a lower cost than (A, B)

Shortest Path From A to G



VISITING F

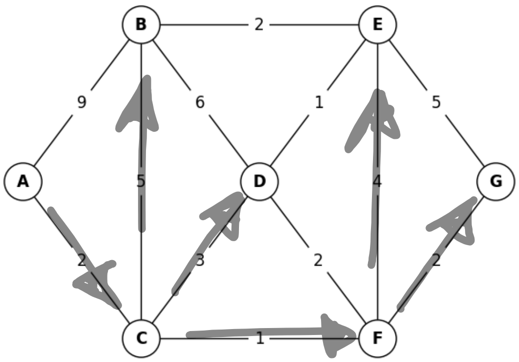
	A	B	C	D	E	F	G
Visited?	X		X			X	
Cost	0	C: 7	A: 2	C: 5	F: 7	C: 3	F: 5

Notice: D's predecessor is unchanged.

In doing so, we ignore the new path (through F) that we examine while visting F

- some path from A to F (cost 3)
- path from F to D (cost 2)

Shortest Path From A to G



"VISITING" G

	A	B	C	D	E	F	G
Visited?	X		X			X	
Cost	0	C:7	A:2	C:5	F:7	C:3	F:5

MOVE BACKWARDS ALONG PREDECESSORS TO FIND SHORTEST PATH

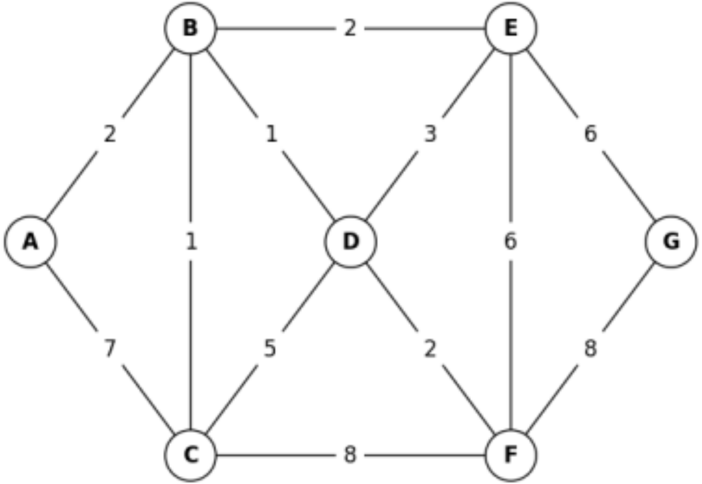
G ← F ← C ← A

How should this look on our HW / exam?

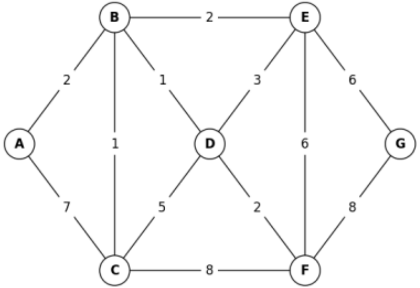
iteration	node visited	A	B	C	D	E	F	G
0	A	start:0	A: 9	A: 2	none	none	none	none
1	C	start:0	C: 7	A: 2	C: 5	none	C: 3	none
2	F	start:0	C: 7	A: 2	C: 5	F: 7	C: 3	F: 5

The path with min weight is: G $\leftarrow$ F $\leftarrow$ C $\leftarrow$ A

In Class Activity:



Using Dijkstra's algorithm, find the shortest path from node A to G. Please provide a table which shows the path weight and predecessor from A to every node, labelling the visited node at each step.



Iteration	Node Visited	A	B	C	D	E	F	G

Full solution to this problem available in "Dijkstra Example".
It includes step-by-step discussion:

- Continue algorithm: visit node with min cost among unvisited: C has cost 3
- ignore path A $\rightarrow$ C $\rightarrow$ D with cost 8 (previous path A $\rightarrow$ B $\rightarrow$ D had cost 3)
- add path A $\rightarrow$ C $\rightarrow$ F with cost 11 (no previous path to F)

iteration	node visited	A	B	C	D	E	F	G
2	C	start:0	A: 2	B: 3	B: 3	B: 4	C: 11	none

pdf available in today today's notes