Directed Acyclic Graphs and Strongly Connected Components

CSE 101: Design and Analysis of Algorithms

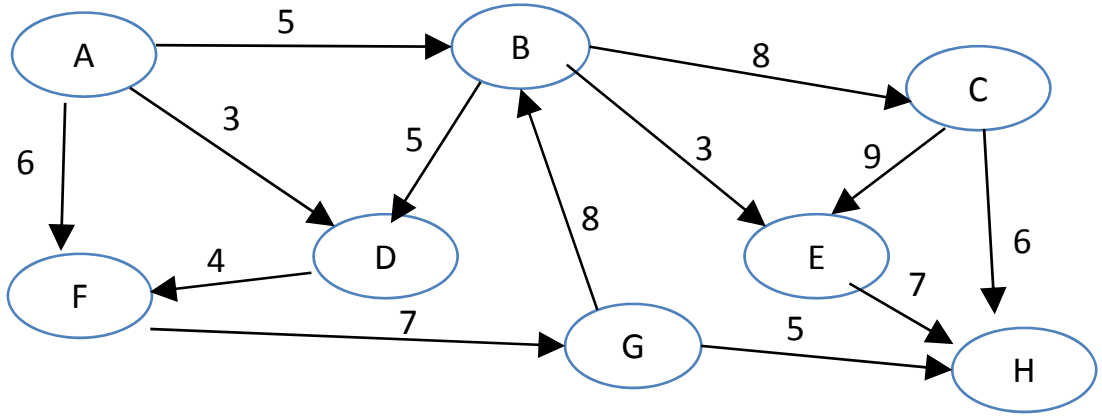
Lecture 3

CSE 101: Design and analysis of algorithms

- Directed acyclic graphs and strongly connected components
  - Reading: Sections 3.3 and 3.4
- Homework 1 due Oct 9, 11:59 PM

Max bandwidth path

- Graph represents network, with edges representing communication links. Edge weights are bandwidth of link.



What is the largest bandwidth of a path from A to H?

Problem statement

- Instance: Directed graph G= (V, E) with positive edge weights w(e), two vertices s, t ∈ V
- Solution type: A path p in G
- Restriction: The path must go from s to t
- Bandwidth of a path

$$\text{BW}(p) = \min_{e \in p} w(e)$$

- Objective: Over all possible paths p between s and t, find the maximum BW(p)

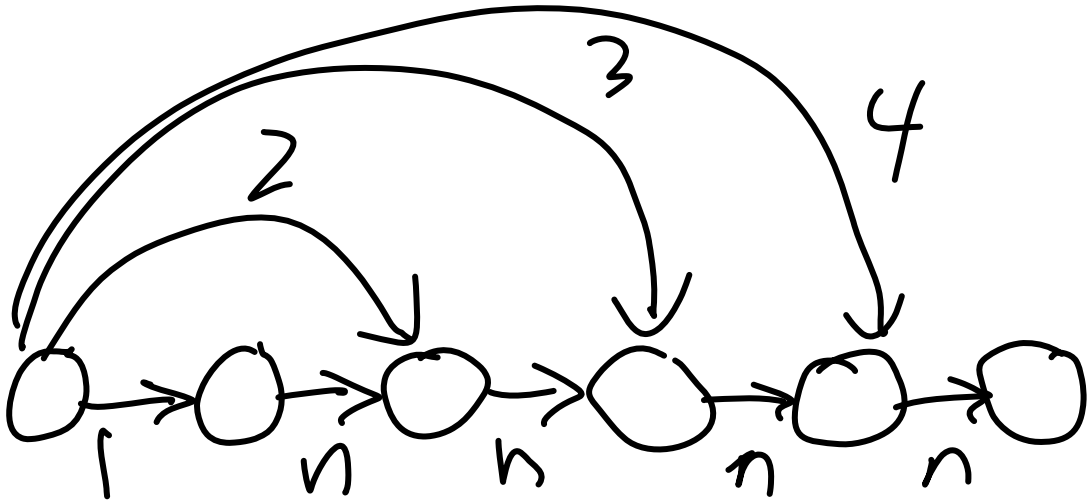
Results from last lecture groups

- Two kinds of ideas
  - Modify an existing algorithm (depth-first search, breadth-first search, Dijkstra's algorithm)
  - Use an existing algorithm (depth-first search) as a sub-routine (possibly modifying the input when you run the algorithm)

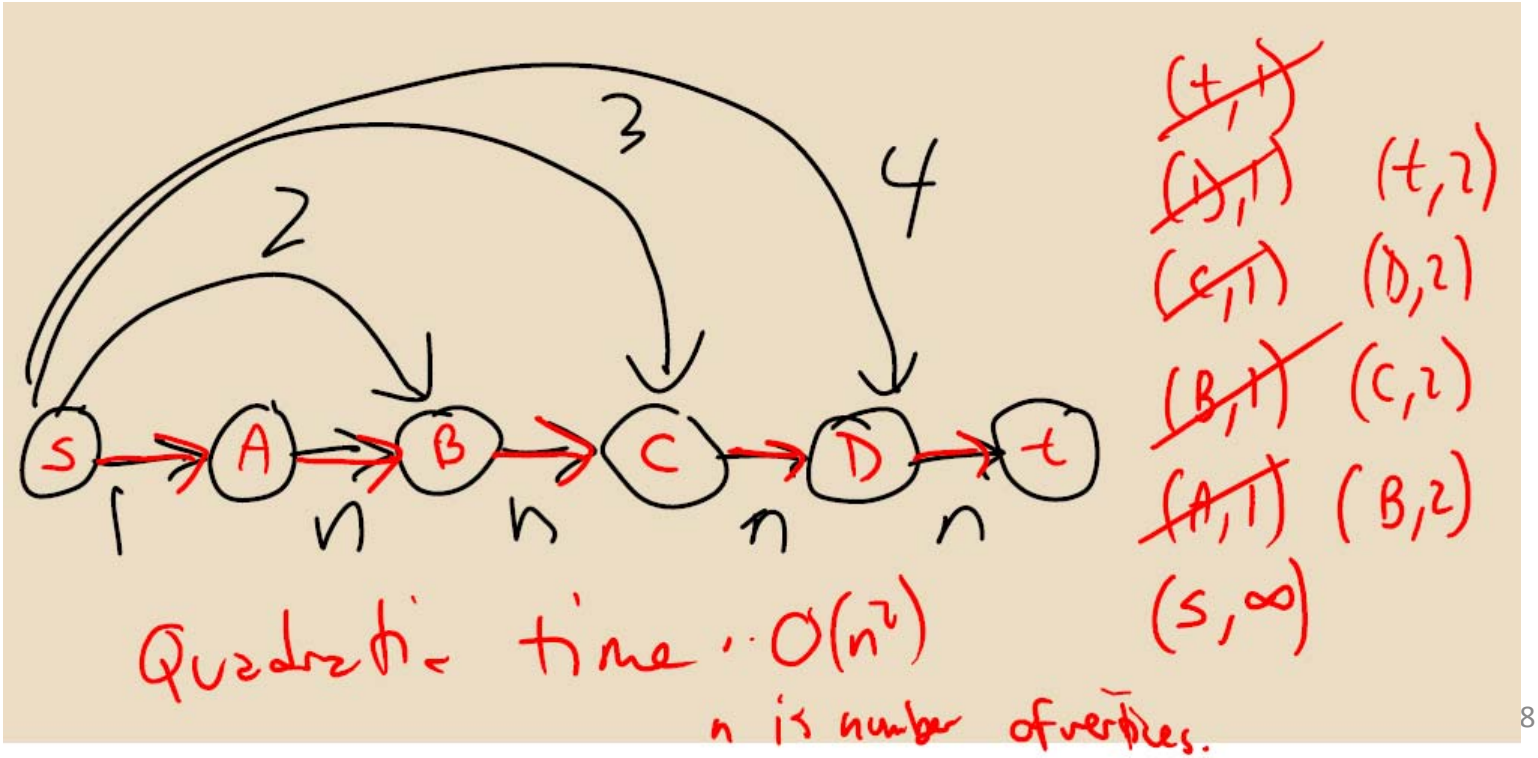
Approaches modifying algorithms

- Use a stack as in depth-first search, but instead of keeping just a vertex, the stack has pairs (V, B), where B is the current bandwidth of A path from s to V. Also, keep an array with the best known bandwidth of paths to each u. When you explore from (V,B), compare the current best bandwidth to each neighbor u with the smaller of B and w((V,u)). If the bandwidth to u improves, update the array and push u with the improved bandwidth.

Example where this method is quadratic time



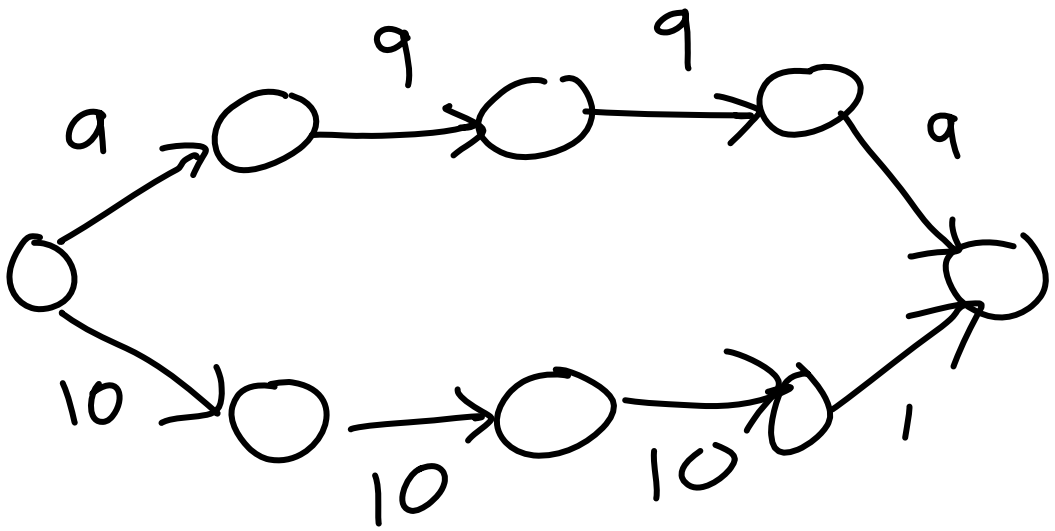
Example where this method is quadratic time



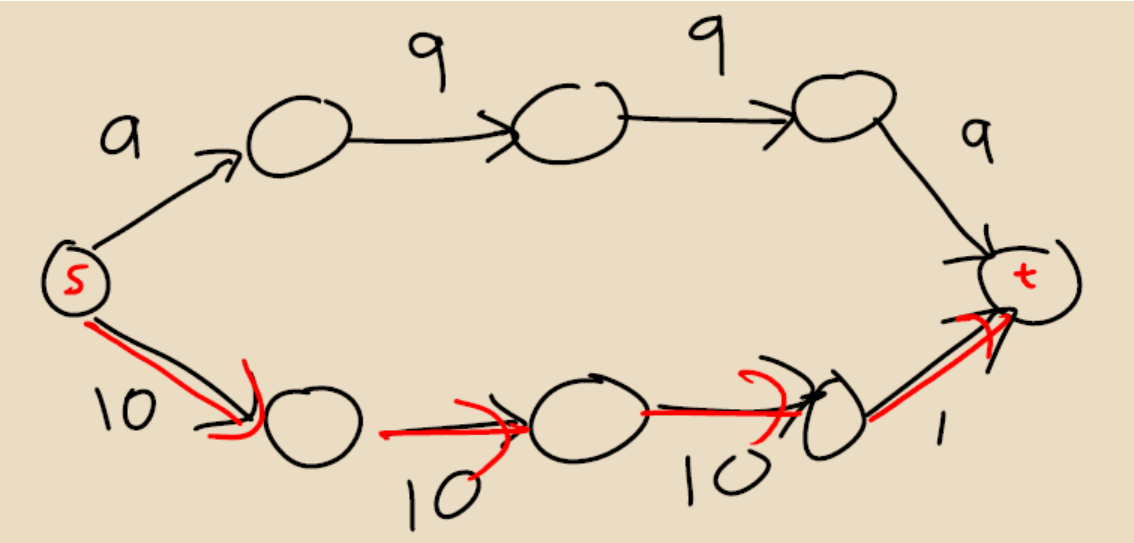
Modifying algorithms

- Use depth-first search, but instead of searching vertices in order of their index, order them by the weight of the edges coming into them

Problematic instance



Problematic instance



Modifying Dijkstra's algorithm

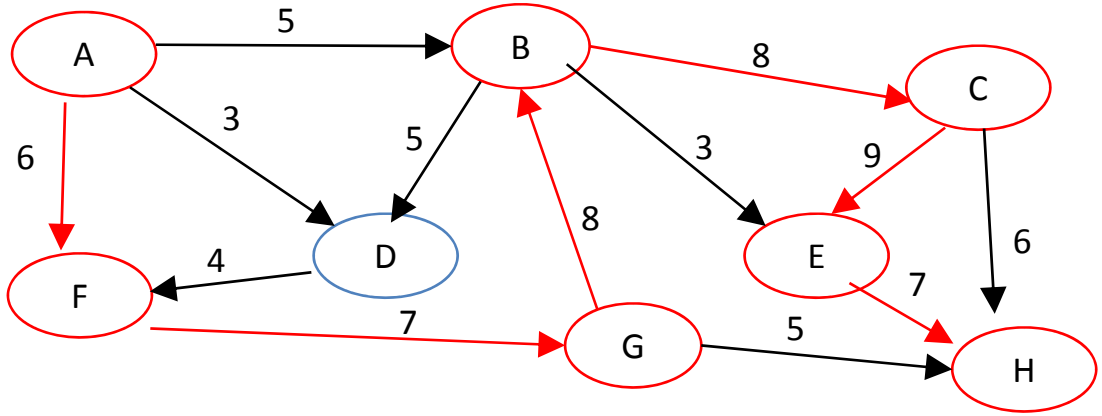
- Use a priority queue as in Dijkstra's algorithm, but using a max-heap keyed by the best bandwidth of a path to v. You explore the highest bandwidth v, and increase the keys for neighbors u with w((v,u)) if higher than the current key.
- This is a good approach, but we'll defer discussion until after reviewing Dijkstra's algorithm (next week)

Using graph search as subroutine

- Keep on removing the smallest weight edge until there is no path from s to t. The weight of the last removed edge is the bandwidth.
- Find some path from s to t using depth-first search. Remove all edges whose weight is at most the smallest weight of an edge in this path and repeat until no path is found. The last edge removed is the bandwidth.

Related approach

- Add edges from highest weight to lowest, stopping when there is a path from s to t



What is the largest bandwidth of a path from A to H?

Reducing to graph search

- These approaches use reductions
- We are using a known algorithm for a related problem to create a new algorithm for a new problem
- Here, the known problem is graph search or graph reachability
- The known algorithms for this problem include depth-first search and breadth-first search
- In a reduction, we map instances of one problem to instances of another. We can then use any known algorithm for that second problem as a sub-routine to create an algorithm for the first.

Graph reachability

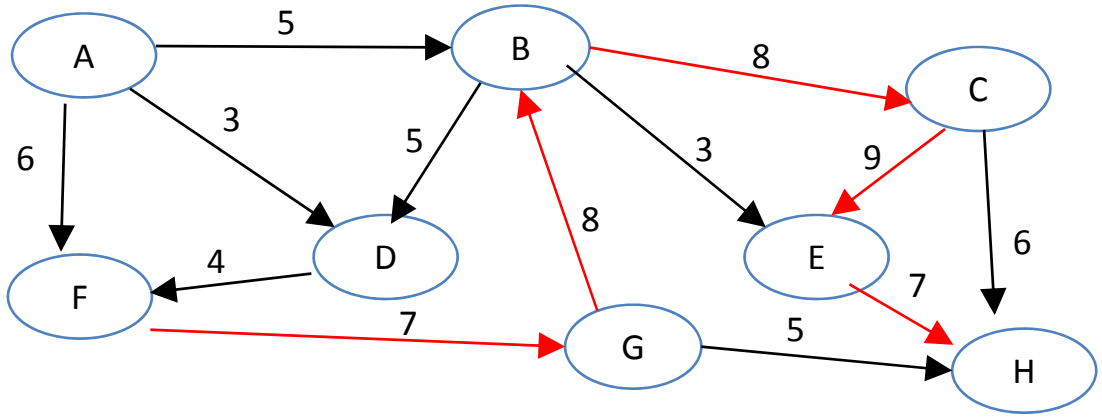
- Given a directed graph G and a start vertex s, produce the set $X \subseteq V$ of all vertices v reachable from s by a directed path in G

Reduction from a decision version

- Reachability is Boolean
  - Yes it is reachable, no it is not
- MaxBandwidth is optimization
  - What is the best bandwidth path?
- To show the connection, let's look at a decision version of max bandwidth path
- Decision version of MaxBandwidth
  - Given G, s, t, and B, is there a path of bandwidth B or better from s to t?

Max bandwidth path

- Say B = 7, and we want to decide whether there is a bandwidth 7 or better path from A to H. Which edges could we use in such a path? Can we use any such edges?



Decision to reachability

- Let $E_B = \{ e : w(e) \geq B \}$
- Lemma: There is a path from s to t of bandwidth at least B if and only if there is a path from s to t in E_B

Decision to reachability

- Let $E_B = \{ e : w(e) \geq B \}$
- Lemma: There is a path from s to t of bandwidth at least B if and only if there is a path from s to t in E_B

- Proof: If p is a path of bandwidth BW ≥ B, then every edge in p must have w(e) ≥ B and so is in E_B. Conversely, if there is a path from s to t with every edge in E_B, the minimum weight edge e in that path must be in E_B, so BW(p)= w(e) ≥ B
- As such, to decide the decision problem, we can use reachability
  - Construct E_B by testing each edge. Then use reachability on s, t, E_B

Reducing optimization to decision

- Suggested approach: if we can test whether the best is at least B, we can find the best value by starting at the largest possible one, and reducing it until we get a yes answer
- Here, possible bandwidths = weights of edges

- In our example, this is the list: 3, 5, 6, 7, 8, 9
- Is there a path of bandwidth 9? If not, then
- Is there a path of bandwidth 8? If not, then
- Is there a path of bandwidth 7? If not, then
- …

Time for this approach

- Let n= |V|, m=|E|
- From previous classes, we know depth-first search, breadth-first search both time O(n+m)
- When we run it on E_B, no worse than running on E, since

$$|E_B| \leq |E|$$

- In the above strategy, how many depth-first search runs do we make in the worst-case?

- What is the total time?

Time for this approach

- Let n= |V|, m=|E|
- From previous classes, we know depth-first search, breadth-first search both time O(n+m)
- When we run it on E_B, no worse than running on E, since

$$|E_B| \leq |E|$$

- In the above strategy, how many depth-first search runs do we make in the worst-case?
  - Each edge might have a different weight, and we might not find a path until we reach the smallest, so we might run depth-first search m times
- What is the total time?
  - Running an $O(n+m)$ algorithm m times means total time $O(m(m+n)) = O(m^2)$

Ideas for improvement

- Is there a better way we could search for the optimal value?

Binary search

- Create sorted array of possible edge weights
  - 3 5 6 7 8 9
- See if there is a path of bandwidth at least the median value
  - Is there a path of bandwidth 6? Yes
- If so, then look in the upper part of the values, if not, then the lower part, always testing the value in the middle
  - 6 7 8 9 Is there a path of bandwidth 8? No
  - 6 7 Is there one of bandwidth 7? No.
  - Therefore, best is 6

Total time for binary search version

- How many depth-first search runs do we need in this version, in the worst case?

- What is the total time of the algorithm?

Total time for binary search version

- How many depth-first search runs do we need in this version, in the worst case?
  - log m runs total = O(log n) runs
- What is the total time of the algorithm?
  - Sorting array: O(m log n) with mergesort
  - O(log n) runs of depth-first search at O(n+m) time per run = O((n+m) log n) time
  - Total: O((n+m) log n)

Differences

- F is stack, last-found is first explored
  - Depth-first search
- F is queue, first found is first explored
  - Breadth-first search
- F is priority queue based on total length
  - Dijkstra's algorithm

Graph reachability

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F:
      move u from U to F.
  Move v from F to X.

Return X.

X: explored
F: frontier (reached but have not yet explored)
U: unreached

Graph reachability

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F:
      move u from U to F.
  Move v from F to X.

Return X.

Let's make F a stack and see what types of problems we can solve.

The book implements a stack using a recursive algorithm:

```
procedure explore(G = (V,E), s)
  visited(s)=true
  for each edge (s,u):
    if not visited(u):
      explore(G,u)
```

Graph reachability

procedure explore (G: directed graph, s: vertex)

```
Initialize F is a stack
push(s,F)
While F is not empty:
  v = head(F)
  if visited(v):
    pop(F)
  For each neighbor u of v:
    If not visited(u)
      push(u,F)
  visited(v) = True
```

Let's make F a stack and see what types of problems we can solve.

The book implements a stack using a recursive algorithm:

```
procedure explore(G = (V,E), s)
  visited(s)=true
  for each edge (s,u):
    if not visited(u):
      explore(G,u)
```

Explore

- Explore takes as input
  - A graph G (directed or undirected) and an initial vertex s
- The output is an array visited such that visited(u) is true if and only if u is reachable from s for all vertices $u \in V$

Keeping track of paths

- We only get information about if there is a path from s to another vertex. What if we want to know what those paths are?

Keeping track of paths

procedure explore (G: directed graph, s: vertex)

Initialize F is a stack
push(s,F)
prev(s) = null
While F is not empty:
  v = **head**(F)
  if visited(v):
    pop(F)
  For each neighbor u of v:
    If not visited(u)
      push(u,F)
      prev(u) = v
  visited(v) = **True**

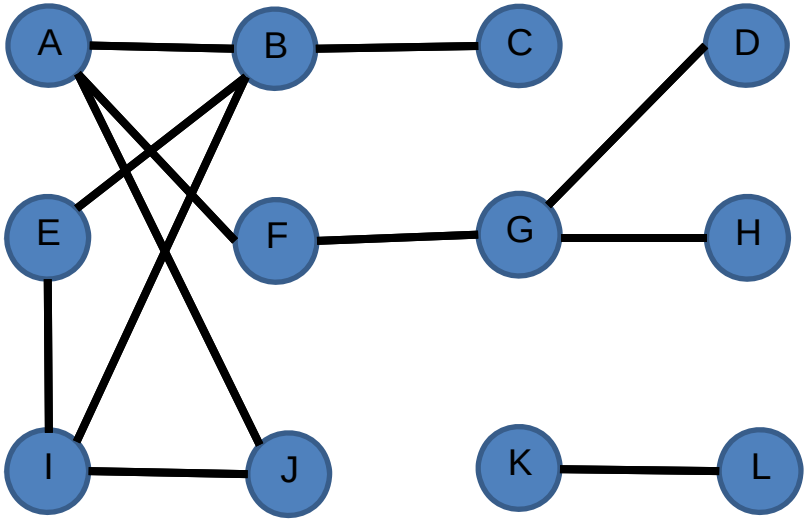
We only get information about if there is a path from s to another vertex. What if we want to know what those paths are?

We can include another array of information. Set prev(u) to be the "parent" of u in the depth-first search output tree

```
procedure explore(G = (V,E), s)
  visited(s)=true
  for each edge (s,u):
    if not visited(u):
      prev(u) = s
      explore(G,u)
```

Keeping track of paths

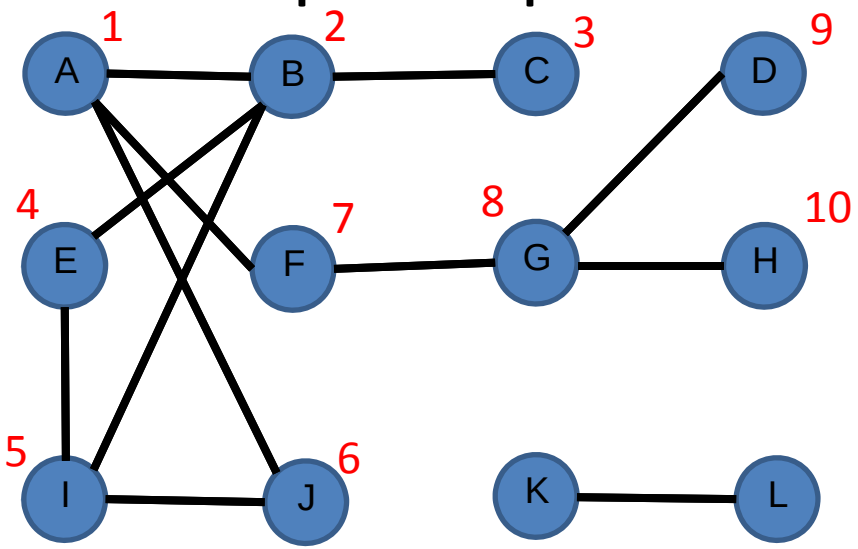
- Example: explore A



Note: when we do graph examples in this class, by default, if a vertex has more than one neighbor, then in the adjacency list, the neighbors are ordered alphabetically unless stated otherwise.

Keeping track of paths

- Example: explore A

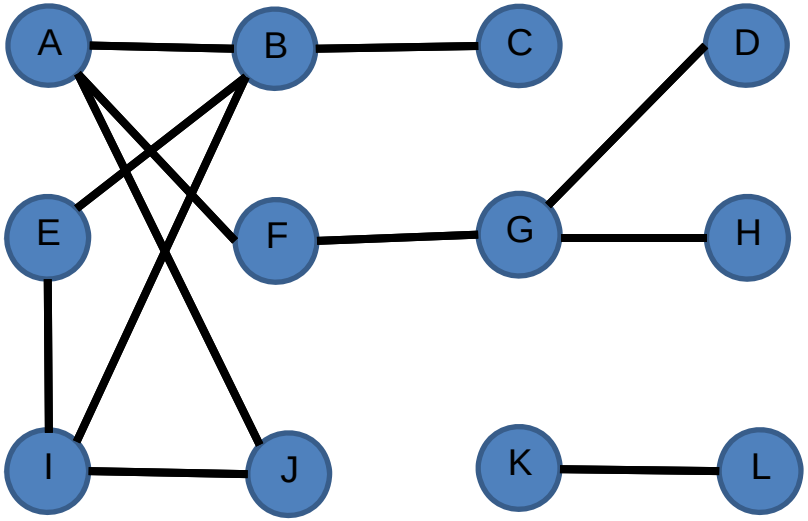


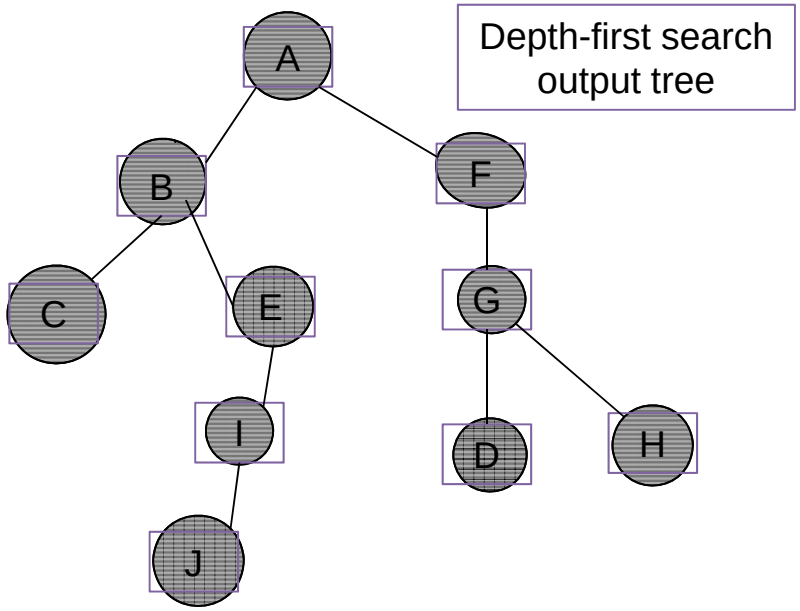
Note: when we do graph examples in this class, by default, if a vertex has more than one neighbor, then in the adjacency list, the neighbors are ordered alphabetically unless stated otherwise.

1 2 3 9 4 7 8 10 5 6
A B C D E F G H I J K L
prev: null A B G B A F G E I

Keeping track of paths

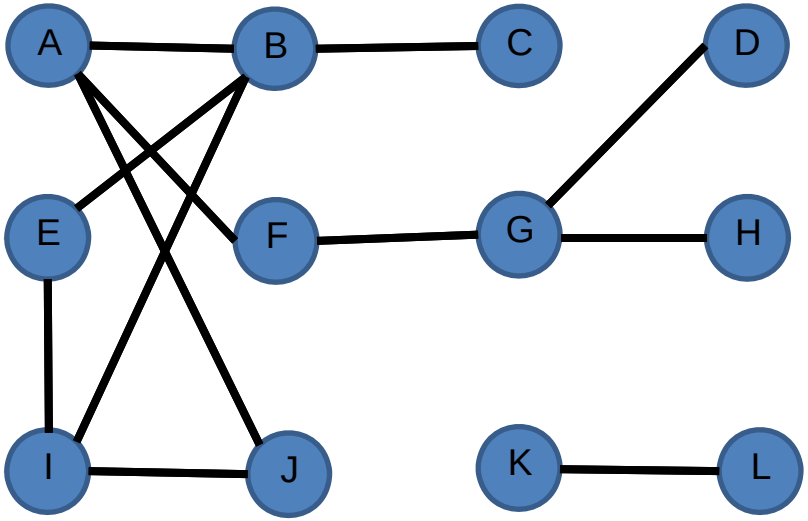
- Example: explore A

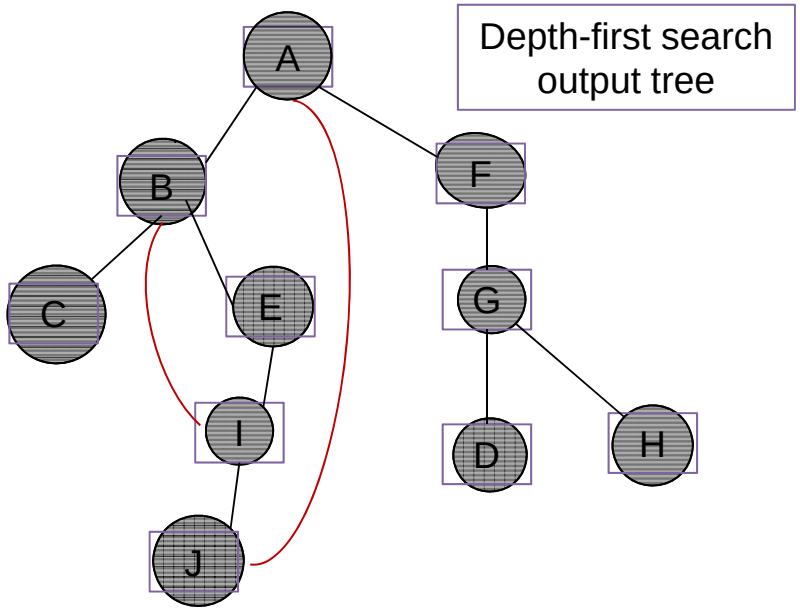




Keeping track of paths

- Example: explore A





Back edges

Back edges in undirected graphs

- Definition: back edges in an undirected graph G that has been explored are edges in G that are not in the depth-first search tree of G
- What do back edges tell you about the graph?

- How do you know if an edge is a back edge?

Back edges in undirected graphs

- Definition: back edges in an undirected graph G that has been explored are edges in G that are not in the depth-first search tree of G
- What do back edges tell you about the graph?
  - Back edges tell you if there are cycles
- How do you know if an edge is a back edge?
  - Each edge (u,v) is a back edge if prev(u)≠v and prev(v)≠u

Cycle edges

- Notice that removing a back edge will not disconnect the graph
- In fact, removing an edge that **is in** a cycle will **not disconnect an undirected graph** and removing an edge that **is not in** a cycle will **disconnect an undirected graph**

Connected undirected graphs

- Definition: An undirected graph G is connected if for every pair of vertices v,u in G, there exists a path from v to u

- explore only reaches one connected component of the graph, namely the set of vertices reachable from s
- To examine the rest of the graph, we need to restart explore on a vertex that has not been visited

Depth first search

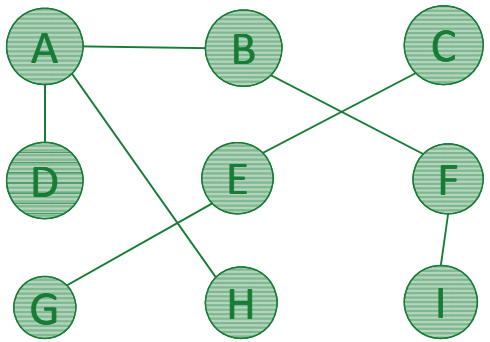
```
procedure explore (G: directed graph, s: vertex)

  Initialize F is a stack
  push(s,F)
  prev(s) = null
  While F is not empty:
    v = head(F)
    if visited(v):
      pop(F)
    For each neighbor u of v:
      If not visited(u)
        push(u,F)
        prev(u) = v
    visited(v) = True
    component(v) = cc
```

```
procedure DFS (G)
  cc = 0
  for each vertex v:
    visited(v) = false
  for each vertex v:
    if not visited(v):
      cc++
      explore(G,v)
```

```
procedure explore(G = (V,E), s)
  visited(s)=true
  for each edge (s,u):
    if not visited(u):
      prev(u) = s
      explore(G,u)
```

Depth first search

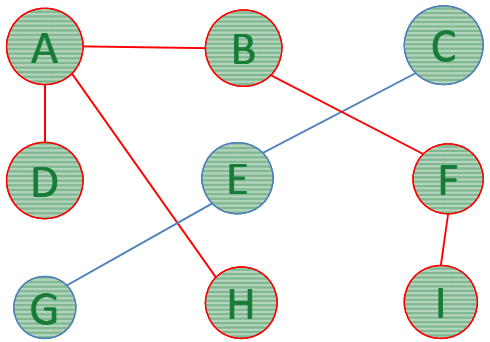
- Example




```
procedure DFS (G)
  cc = 0
  for each vertex v:
    visited(v) = false
  for each vertex v:
    if not visited(v):
      cc++
      explore(G,v)
```

Depth first search

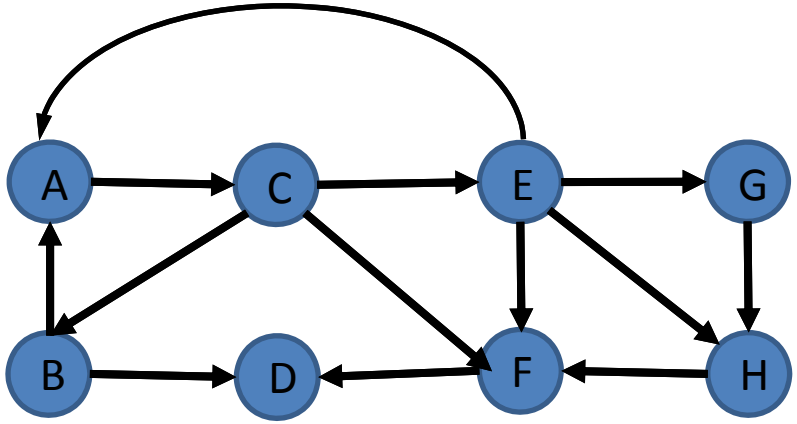
- Example



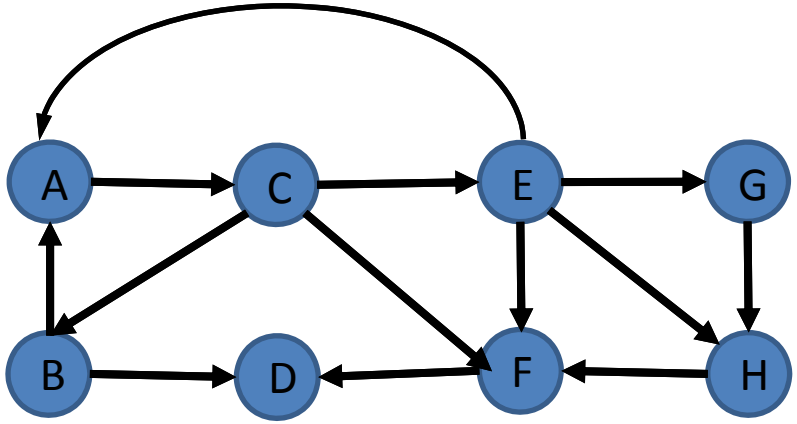

```
procedure DFS (G)
  cc = 0
  for each vertex v:
    visited(v) = false
  for each vertex v:
    if not visited(v):
      cc++
      explore(G,v)
```

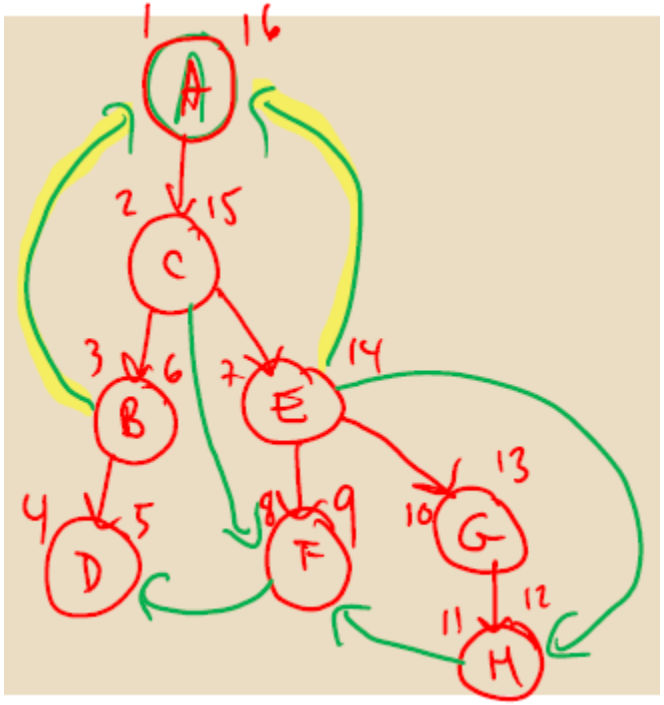
	A	B	C	D	E	F	G	H	I
cc:	1	1	2	1	2	1	2	1	1
prev:	null	A	null	A	C	B	E	A	F

Depth-first search on directed graphs



Depth-first search on directed graphs





Pre and post numbers

- Pre and post numbers assign 2 integers to each vertex
- These correspond to the steps of when that vertex enters the stack and leaves the stack

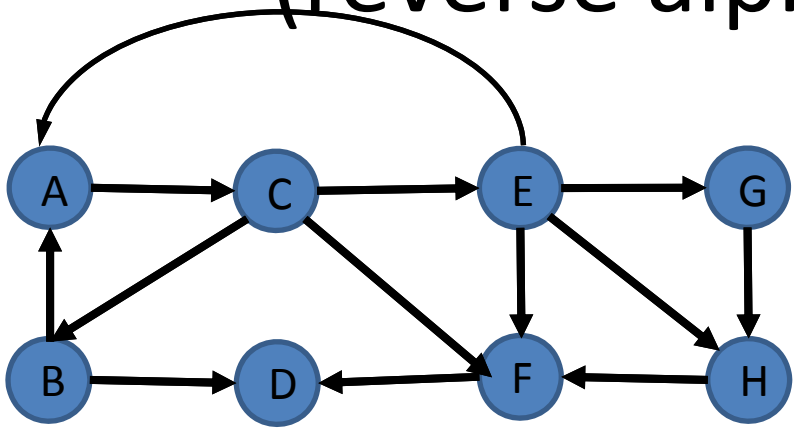
```
procedure DFS (G)
  cc = 0
  clock = 1
  for each vertex v:
    visited(v) = false
  for each vertex v:
    if not visited(v):
      cc++
      explore(G,v)
```

```
procedure explore(G = (V,E), s)
  visited(s)=true
  component(s)=cc
  previsit(s)
  for each edge (s,u):
    if not visited(u):
      prev(u) = s
      explore(G,u)
  postvisit(s)
```

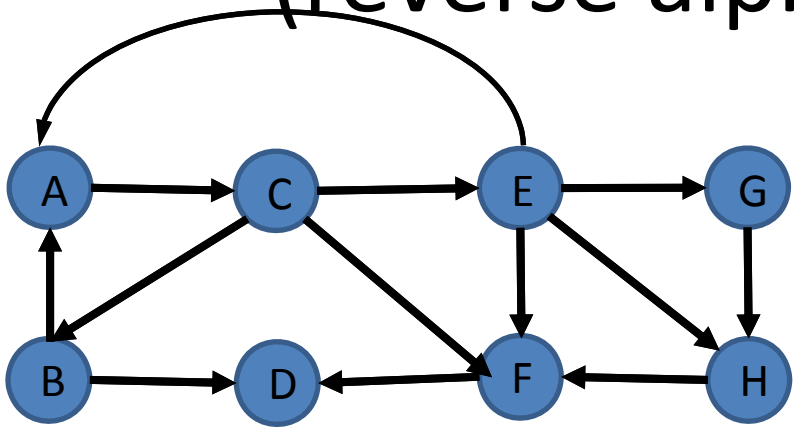
```
procedure previsit(v)
  pre(v)=clock
  clock++
```

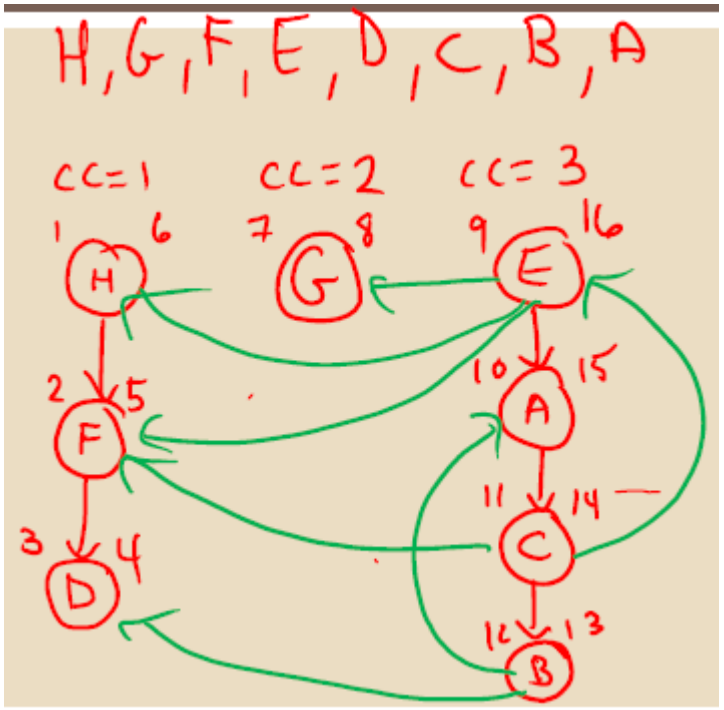
```
procedure postvisit(v)
  post(v)=clock
  clock++
```

Depth-first search on directed graphs (reverse alphabetical order)



Depth-first search on directed graphs (reverse alphabetical order)





Edge types (directed graph)

- Tree edge: solid edge included in the depth-first search output tree
- Back edge: leads to an ancestor
- Forward edge: leads to a descendent
- Cross edge: leads to neither ancestor or descendent

- Note that **back edge** is slightly different in directed and undirected graphs

Edge types and pre/post numbers

- The different types of edges can be determined from the pre/post numbers for the edge (u, v)
  - (u, v) is a tree/forward edge then
    $pre(u) < pre(v) < post(v) < post(u)$
  - (u, v) is a back edge then
    $pre(v) < pre(u) < post(u) < post(v)$
  - (u, v) is a cross edge then
    $pre(v) < post(v) < pre(u) < post(u)$

Next lecture

- Strongly connected components and breadth-first search
  - Reading: Sections 3.4, 4.1, 4.2, and 4.3