CS3000: Algorithms & Data
Jonathan Ullman

Lecture 19:

- Midterm II Review

Nov 13, 2018

Topics to Review

- Key Graph Definitions / Properties
  - Directed/Undirected
  - Weighted/Unweighted
  - Trees, DAGs
  - Paths, Cycles
  - Connected Components, Strongly Connected Components

Graphs: Key Definitions

- **Definition:** A directed graph $G = (V, E)$
  - V is the set of nodes/vertices
  - $E \subseteq V \times V$ is the set of edges
  - An edge is an ordered $e = (u, v)$ "from u to v"

$|V| = n$

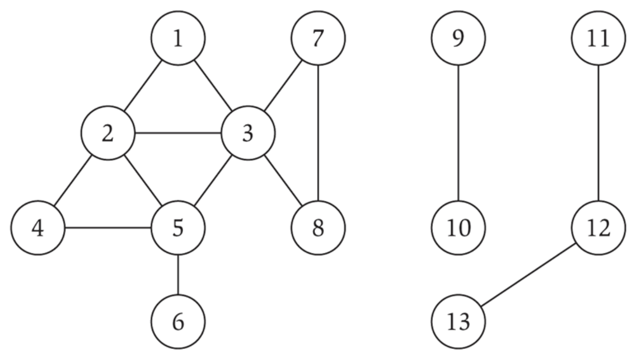
$|E| = m$

undirected $m \in [0, \binom{n}{2}]$

$m \in [0, n(n-1)]$ directed

- **Definition:** An undirected graph $G = (V, E)$
  - Edges are unordered $e = (u, v)$ "between u and v"

- **Simple Graph:**
  - No duplicate edges
  - No self-loops $e = (u, u)$



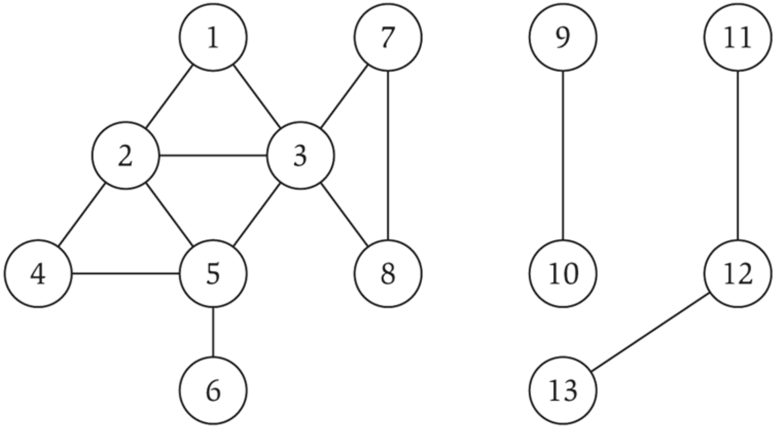
Paths/Connectivity

- A path is a sequence of consecutive edges in E
  - $P = \{(u, w_1), (w_1, w_2), (w_2, w_3), \dots, (w_{k-1}, v)\}$
  - $P = u - w_1 - w_2 - w_3 - \cdots - w_{k-1} - v$
  - The length of the path is the # of edges

- An undirected graph is connected if for every two vertices $u, v \in V$, there is a path from u to v
- A directed graph is strongly connected if for every two vertices $u, v \in V$, there are paths from u to v and from v to u

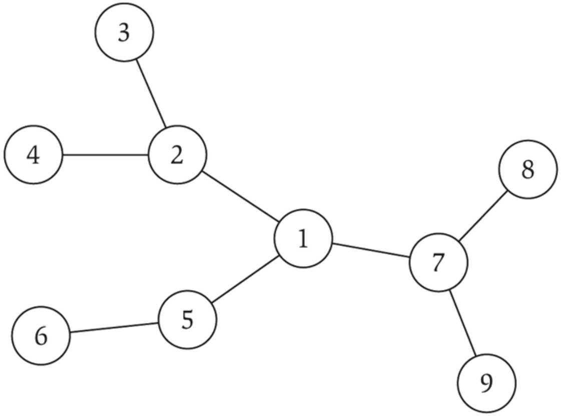
Cycles

- A cycle is a path $v_1 - v_2 - \cdots - v_k - v_1$ where $k \geq 3$ and $v_1, \ldots, v_k$ are distinct



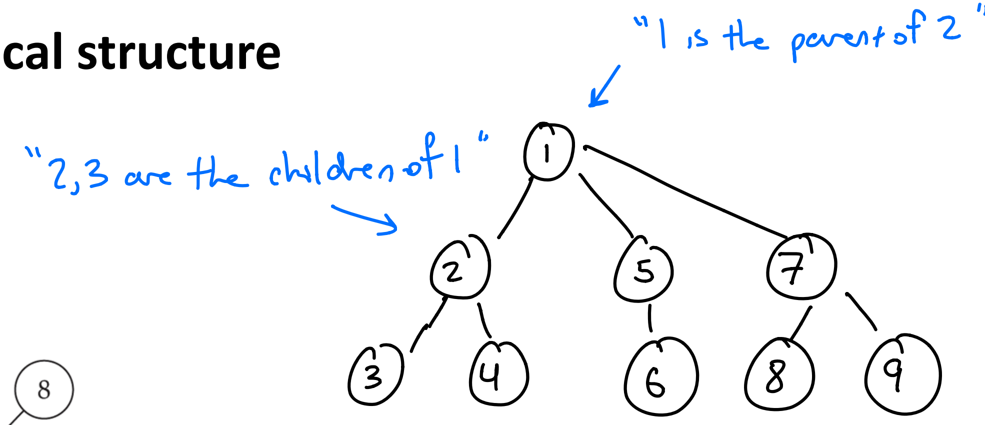
Trees

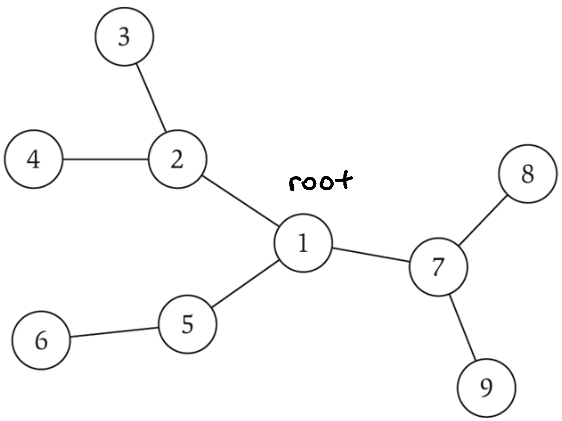
- A simple undirected graph G is a tree if:
  - G is connected
  - G contains no cycles
- **Theorem:** any two of the following implies the third
  - G is connected
  - G contains no cycles
  - G has $= n - 1$ edges



Trees

- **Rooted tree:** choose a root node $\boldsymbol{r}$ and orient edges away from $\boldsymbol{r}$
  - Models **hierarchical structure**





Topics to Review

- Graph Representations
  - Adjacency Matrix
  - Adjacency List] All algorithms we study use adjacency list

Adjacency-Matrix Representation

- The adjacency matrix of a graph $G = (V, E)$ with n nodes is the matrix $A[1{:}n, 1{:}n]$ where

$$A[i,j] = \begin{cases} 1 & (i,j) \in E \\ 0 & (i,j) \notin E \end{cases}$$

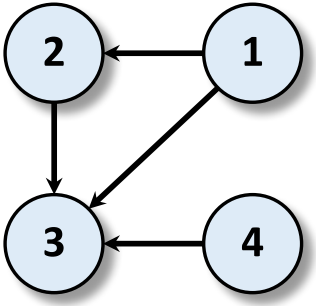
A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

Cost

Space: $\Theta(n^2)$

Lookup (u,v): $\Theta(1)$ time

List Neighbors of u: $\Theta(n)$ time



Adjacency Lists (Directed)

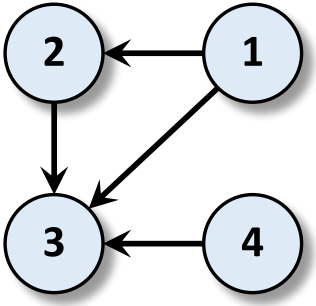
- The adjacency list of a vertex $v \in V$ are the lists
  - $A_{out}[v]$ of all u s.t. $(v, u) \in E$
  - $A_{in}[v]$ of all u s.t. $(u, v) \in E$

$A_{out}[1] = \{2,3\}$ $A_{in}[1] = \{\ \}$

$A_{out}[2] = \{3\}$ $A_{in}[2] = \{1\}$

$A_{out}[3] = \{\ \}$ $A_{in}[3] = \{1,2,4\}$

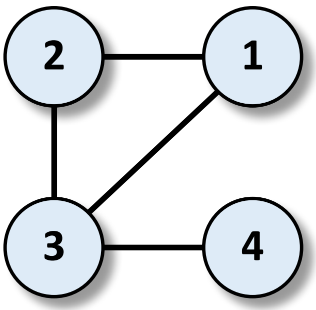
$A_{out}[4] = \{3\}$ $A_{in}[4] = \{\ \}$



Adjacency-List Representation

- The adjacency list of a vertex $v \in V$ is the list $A[v]$ of all the neighbors of v

$$A[1] = \{2,3\}$$
$$A[2] = \{1,3\}$$
$$A[3] = \{1,2,4\}$$
$$A[4] = \{3\}$$



<u>Cost</u>

Space: $\Theta(n + m)$

Lookup (u,v): $\Theta(\deg(u) + 1)$ time

List Neighbors of u: $\Theta(\deg(u) + 1)$ time

Topics to Review

- Finding (short) paths in graphs
  - BFS for finding:
    - Connected components
    - Strongly connected components
    - Shortest paths in unweighted graphs (i.e. fewest hops)
  - Dijkstra's algorithm for finding:
    - Shortest paths in graphs with non-negative lengths
  - Bellman-Ford algorithm for finding:
    - Shortest paths in graphs with negative lengths (no neg cycles)
    - Negative cycles if they exist
  - Structural properties of shortest paths
    - Dynamic programming $\forall (u,v) \in E, \quad d(s,v) \leq d(s,u) + \ell(u,v)$
    - Shortest path trees

BFS

- Informal Description: start at s, find all neighbors of s, find all neighbors of neighbors of s, …

- BFS Algorithm:
  - $L_0 = \{s\}$
  - $L_1 =$ all neighbors of L_0
  - $L_2 =$ all neighbors of L_1 that are not in L_0, L_1
  - …
  - $L_d =$ all neighbors of L_{d-1} that are not in $L_0, \ldots, L_{d-1}$
  - Stop when L_{d+1} is empty.

Breadth-First Search Implementation

```
BFS(G = (V,E), s):
  Let found[v] ← false ∀v, found[s] ← true
  Let layer[v] ← ∞ ∀v, layer[s] ← 0
  Let i ← 0, L_0 = {s}, T ← ∅

  While (L_i is not empty):
    Initialize new layer L_{i+1}
    For (u in L_i):
      For ((u,v) in E):
        If (found[v] = false):
          found[v] ← true, layer[v] ← i+1
          Add (u,v) to T and add v to L_{i+1}
    i ← i+1
```

Implementing Dijkstra

```
Dijkstra(G = (V,E,{ℓ(e)}, s):
  d[s] ← 0, d[u] ← ∞ for every u != s
  parent[u] ←⊥ for every u
  Q ← V                    // Q holds the unexplored nodes

  While (Q is not empty):
    u ← argmin_{w∈Q} d[w]   //Find closest unexplored
    Remove u from Q

    // Update the neighbors of u
    For ((u,v) in E):
      If (d[v] > d[u] + ℓ(u,v)):
        d[v] ← d[u] + ℓ(u,v)
        parent[v] ← u

  Return (d, parent)
```

Recurrence

- **Subproblems:** $\mathrm{OPT}(v, j)$ is the length of the shortest $s \leadsto v$ path with at most j hops
- **Case u:** (u, v) is final edge on the shortest $s \leadsto v$ path with at most j hops

Recurrence:

$$\mathrm{OPT}(v, j) = \min\left\{\mathrm{OPT}(v, i-1), \min_{(u,v) \in E}\{\mathrm{OPT}(u, i-1) + \ell_{u,v}\}\right\}$$

$\mathrm{OPT}(s, j) = 0$ for every j

$\mathrm{OPT}(v, 0) = \infty$ for every v

Implementation (Bottom Up)

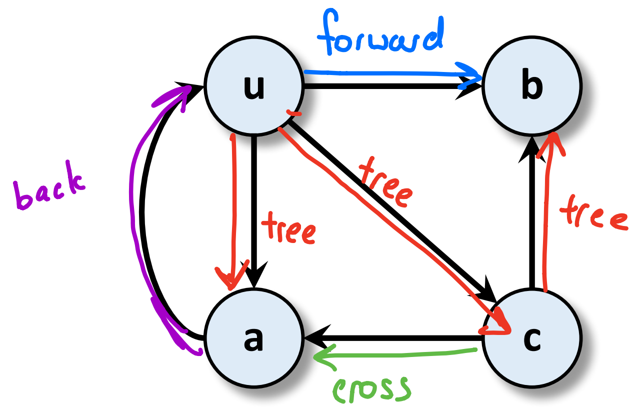
```
Shortest-Path(G, s)
   foreach node v ∈ V
      M[0,v] ← ∞
      P[0,v] ← ϕ
   M[0,s] ← 0

   for i = 1 to n-1
      foreach node v ∈ V
        M[i,v] ← M[i-1,v]
        P[i,v] ← P[i-1,v]
        foreach edge (v, w) ∈ E
           if (M[i-1,w] + ℓwv < M[i,v])
              M[i,v] ← M[i-1,w] + ℓwv
              P[i,v] ← w
```

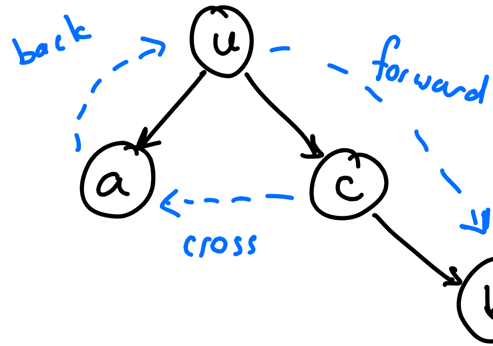
Topics to Review

- Depth-First Search
  - Types of edges (tree, forward, backward, cross)
  - Post-ordering (Pre-ordering)
- Topological Sort
  - Fast algorithm using DFS
- Other graph algorithms
  - 2-coloring

Depth-First Search

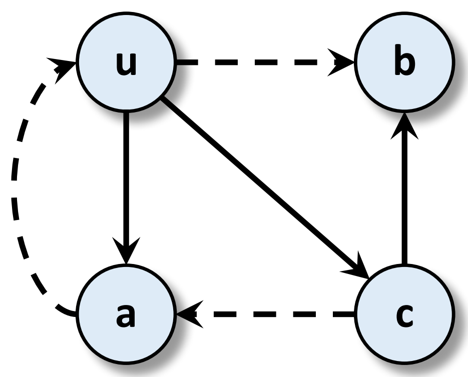
```
G = (V,E) is a graph
explored[u] = 0 ∀u

DFS(u):
  explored[u] = 1

  for ((u,v) in E):
    if (explored[v]=0):
      parent[v] = u
      DFS(v)
```

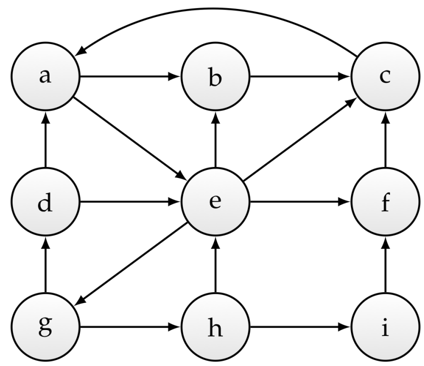




Depth-First Search

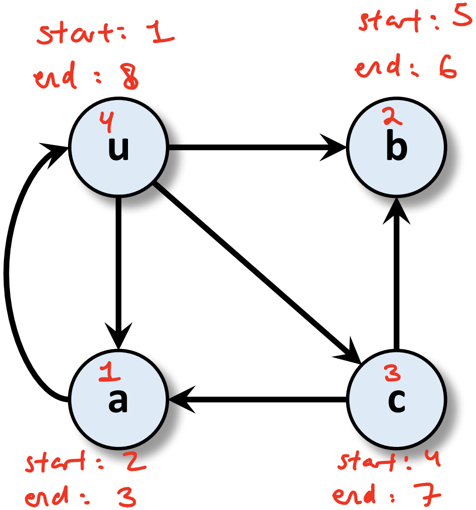
- **Fact:** The parent-child edges form a (directed) tree
- **Each edge has a type:**
  - **Tree edges:** $(u, a), (u, c), (c, b)$
    - These are the edges that explore new nodes
  - **Forward edges:** (u, b)
    - Ancestor to descendant
  - **Backward edges:** (a, u)
    - Descendant to ancestor
  - **Cross edges:** (c, a)
    - No ancestral relation




a
b
c
d
e
f
g
h
i

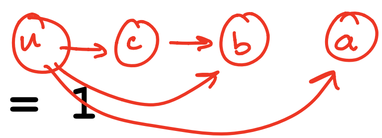
Post-Ordering

```
G = (V,E) is a graph
explored[u] = 0 ∀u

DFS(u):
  explored[u] = 1

  for ((u,v) in E):
    if (explored[v]=0):
      parent[v] = u
      DFS(v)

  post-visit(u)
```



Vertex	Post-Order
u	4
a	1
b	2
c	3

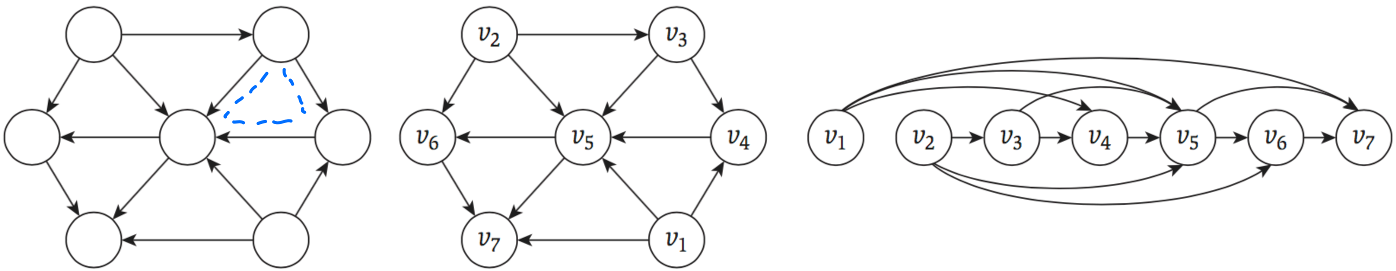
pre-ordering
1
2
4
3



- Maintain a counter **clock**, initially set **clock = 1**
- **post-visit(u):**
  set postorder[u]=clock, clock=clock+1

Directed Acyclic Graphs (DAGs)

- **DAG:** A directed graph with no directed cycles
- DAGs represent precedence relationships



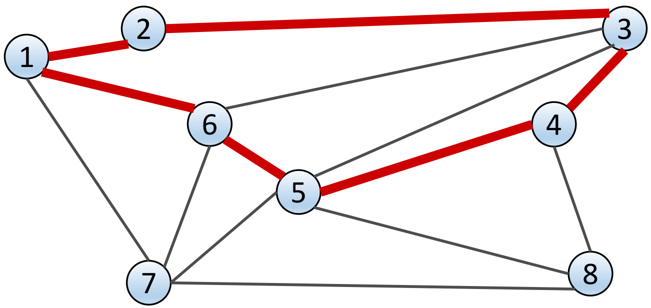
- A **topological ordering** of a directed graph is a labeling of the nodes from $v_1, \ldots, v_n$ so that all edges go "forwards", that is $(v_i, v_j) \in E \Rightarrow j > i$
  - **G has a topological ordering $\Leftrightarrow$ G is a DAG**
  - Reverse of post-order is a topological ordering

Topics to Review

- Minimum Spanning Trees
  - Cut Property / Cycle Property
  - Four Algorithms:
    - Boruvka
    - Prim
    - Kruskal
    - Anti-Kruskal

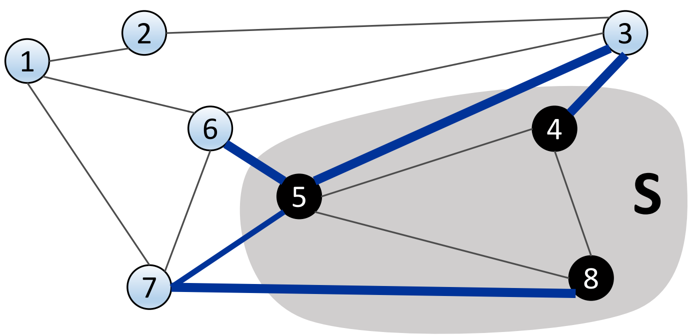
Cycles and Cuts

- Cycle: a set of edges $(v_1, v_2), (v_2, v_3), \dots, (v_k, v_1)$



Cycle C = (1,2),(2,3),(3,4),(4,5),(5,6),(6,1)

- Cut: a subset of nodes S



Cut S = {4, 5, 8}
Cutset = (5,6), (5,7), (3,4), (3,5), (7,8)

Properties of MSTs

- Assuming edge weights are distinct
- Cut Property: Let S be a cut. Let e be the minimum weight edge cut by S. Then the MST T^* contains e
  - We call such an e a safe edge
- Cycle Property: Let C be a cycle. Let e be the maximum weight edge in C. Then the MST T^* does not contain e.
  - We call such an e a useless edge

MST Algorithms

- There are at least four reasonable MST algorithms
  - Borůvka's Algorithm: start with $T = \emptyset$, in each round add cheapest edge out of each connected component
  - Prim's Algorithm: start with some s, at each step add cheapest edge that grows the connected component
  - Kruskal's Algorithm: start with $T = \emptyset$, consider edges in ascending order, adding edges unless they create a cycle
  - Reverse-Kruskal: start with $T = E$, consider edges in descending order, deleting edges unless it disconnects

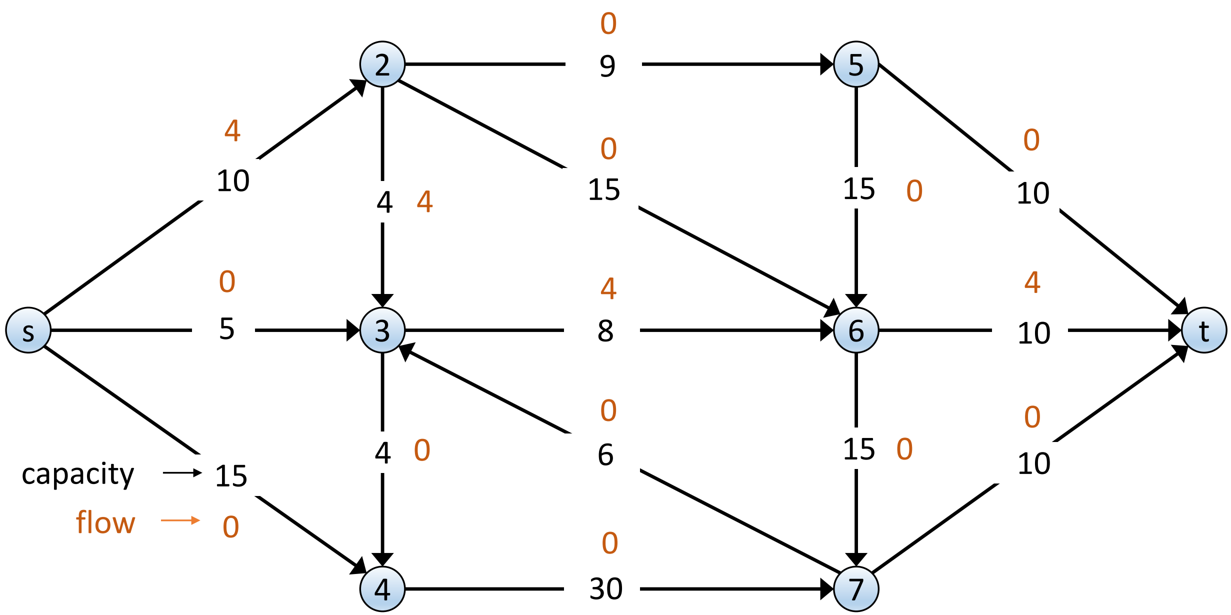
Topics to Review

- Network Flow
  - Definitions (Flows, Cuts, Augmenting Path, Residual Graph)
  - Ford-Fulkerson Algorithm
    - Algorithm
    - Correctness
    - Running time analysis
    - Methods for choosing good augmenting paths (but not proofs)
  - MaxFlow-MinCut Theorem
  - We can compute a max flow in $O(mn)$ time

Flows

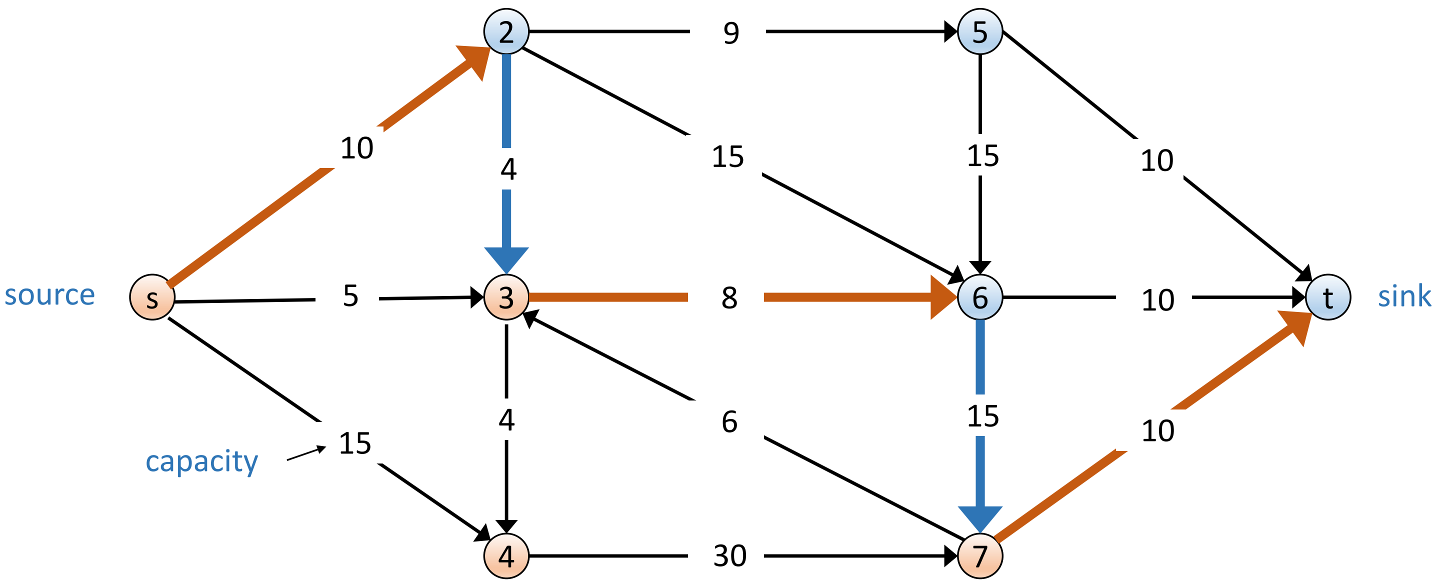
- An s-t flow is a function $f(e)$ such that
  - For every $e \in E, 0 \leq f(e) \leq c(e)$ (capacity)
  - For every $v \in E, \sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ (conservation)

- The value of a flow is $val(f) = \sum_{e \text{ out of } s} f(e)$



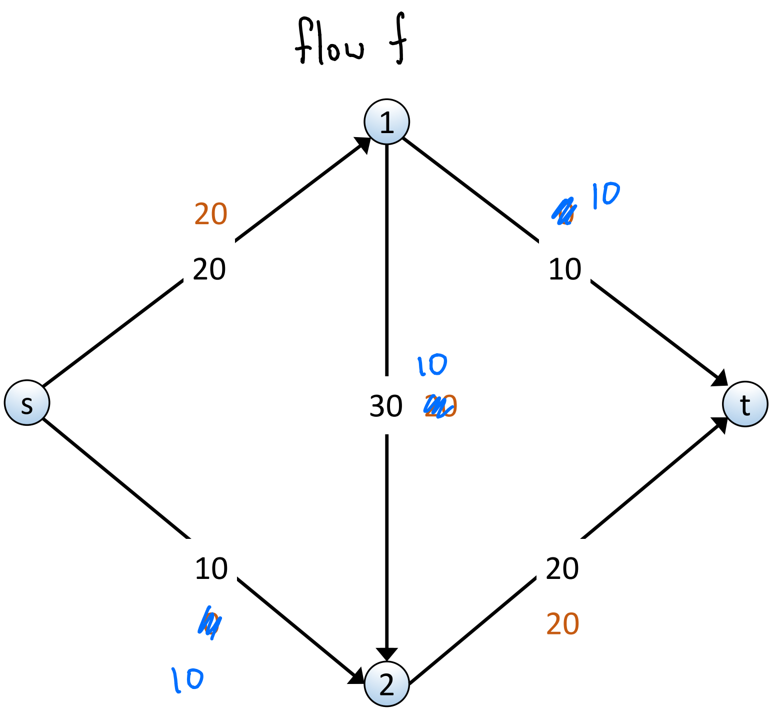
Cuts

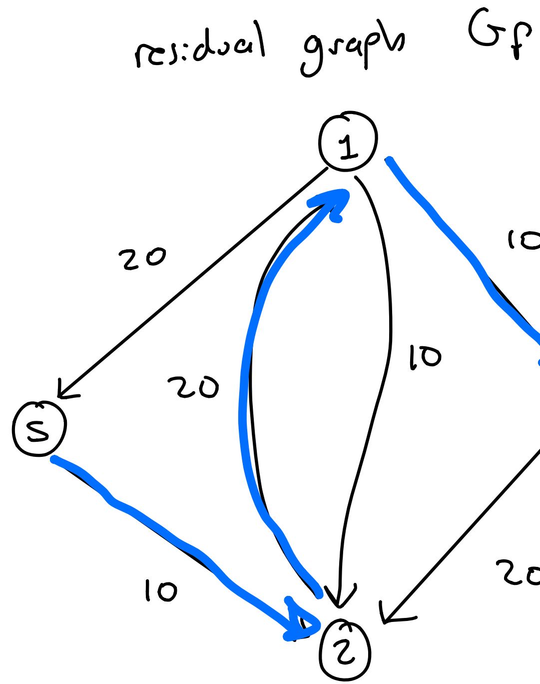
- An s-t cut is a partition (A, B) of V with $s \in A$ and $t \in B$
- The capacity of a cut (A,B) is $cap(A, B) = \sum_{e \text{ out of } A} c(e)$



Ford-Fulkerson Algorithm

- Start with $f(e) = 0$ for all edges $e \in E$
- Find an **augmenting path** P in the **residual graph**
- Repeat until you get stuck





Ford-Fulkerson Algorithm

```
FordFulkerson(G,s,t,{c})
    for e ∈ E: f(e) ← 0
    G_f is the residual graph

    while (there is an s-t path P in G_f)
        f ← Augment(G_f,P)
        update G_f

    return f
```

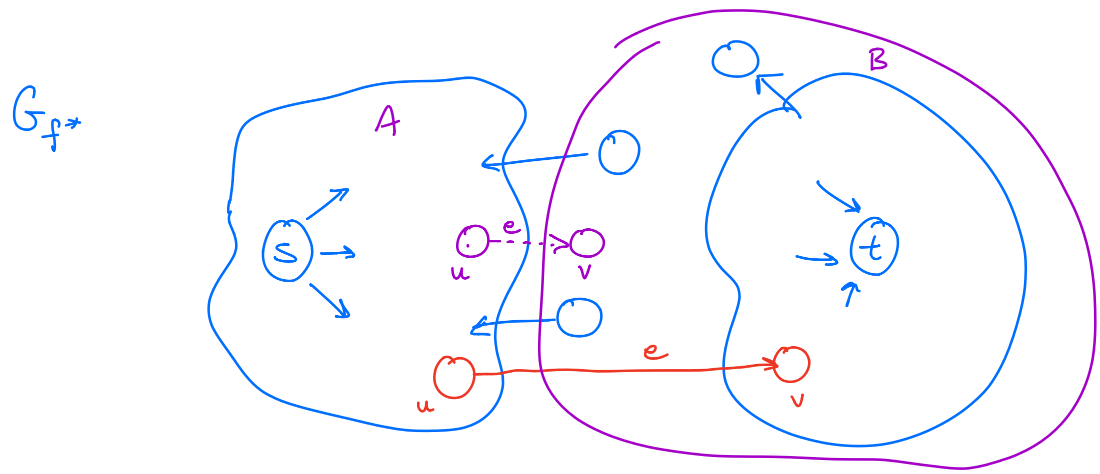
O(m) time per aug path

```
Augment(G_f, P)
    b ← the minimum capacity of an edge in P
    for e ∈ P
        if e ∈ E:   f(e) ← f(e) + b
        else:       f(e) ← f(e) - b
    return f
```

Review Problems

Review Question:

Given a flow network $G = (V, E, s, t, \{c(e)\})$ (integer capacities) and a maximum flow f^*, find all edges $e \in E$ s.t. increasing $c(e)$ by 1 will increase the value of the maximum flow.



How would increasing $c(e)$ by 1 change the residual graph

- If $f^*(e) < c(e)$, then the edge was already in the residual graph
- If $f^*(e) = c(e)$, then increasing capacity by 1 puts e back in the residual graph
  - Increase the max flow iff u is reachable from s, t is reachable from v. $(e = (u, v))$

Pseudocode

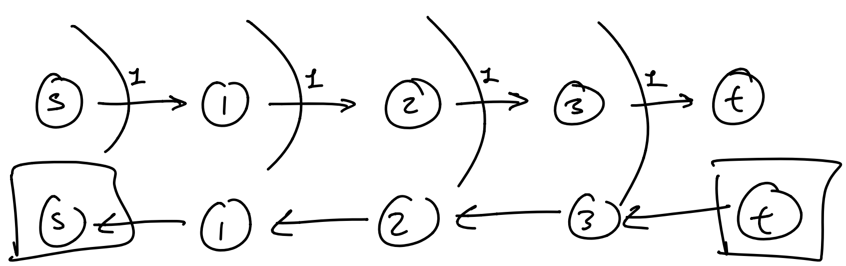
- Let L be all nodes reachable from s in G_f.
- Let R be all nodes reachable from t in G_f. (using edges backwards)
- $S = \emptyset$
- For ($(u,v) \in E$):

    If ($u \in L \wedge v \in R$):

        add (u,v) to S

- Output S

$$\max_{f} val(f) = \min_{(A,B)} cap(A,B)$$



Bonus Review Problem

- Prove the following by induction: in any rooted binary tree, the number of nodes with exactly two children is one less than the number of leaves.

Review Problem #4

- Design an algorithm that takes an undirected $G = (V, E)$, and a pair of nodes s, t and outputs the number of shortest s-t paths in G.

Review Problem #5

- Design an algorithm to find a fattest s-t path in a flow network $G = (V, E, s, t, \{c(e)\})$

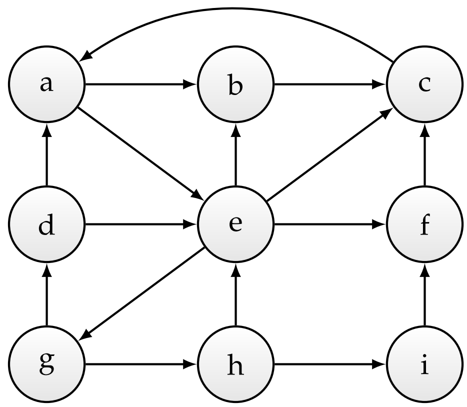
Review Problem #6

- There are n bank accounts $A_1, \ldots, A_n$, and you are given m constraints of the form
  - A_i was closed before A_j opened
  - A_i and A_j were open (at least partially) simultaneously
- Design an algorithm to determine if there are opening and closing times for the accounts that satisfy all constraints

Review Problem #7

- Prove the following by contradiction: if G is an undirected simple graph with $2n$ nodes, and every node has degree $\geq n$, then G is connected.

Problem 1. *DFS and Topological Ordering*



Consider running depth-first search on this graph starting from node a. When there are multiple choices for the next node to visit, go in alphabetical order.

(a) Label every edge as either tree, forward, backward, or cross.

Solution:

(b) Give the post-order numbers of all vertices

Solution:

(c) Is this graph a DAG? Support your answer by either showing a topological ordering or a directed cycle.

Solution: