CS 580
ALGORITHM DESIGN AND ANALYSIS

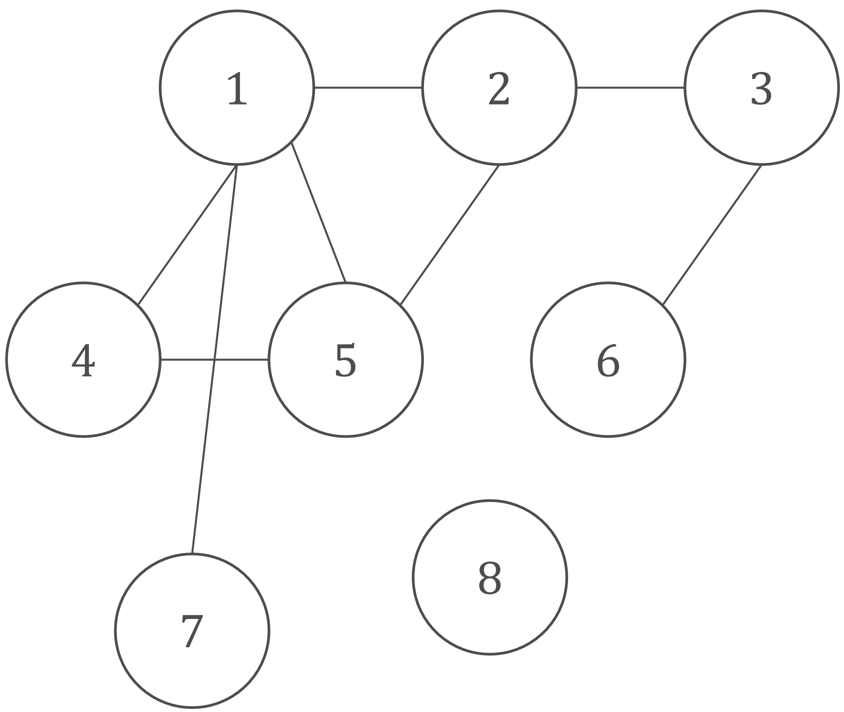
Basics: Graphs
(Chapter 3 in the "Algorithm Design" book)

Vassilis Zikas

BASIC DEFINITIONS: GRAPHS

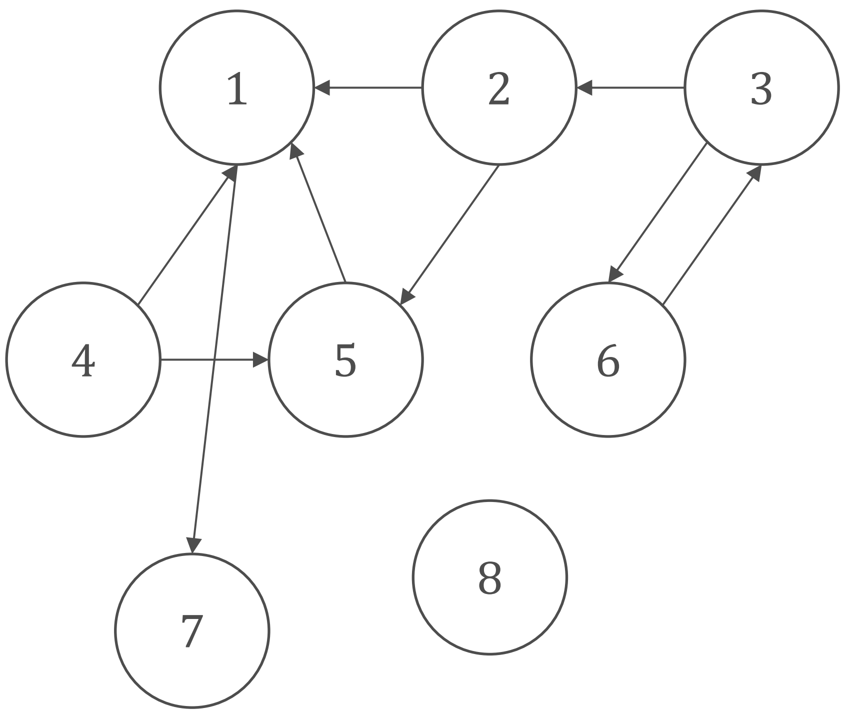
- A graph $G = (V, E)$ consists of:
  - A set V of nodes/vertices
  - A set E of pairs of nodes, the edges
    - $e \in \mathrm{E}$ if $e = (u, v)$ for some $u, v \in \mathrm{V}$
  - Typically $n = |V|$ and $m = |E|$
- A directed graph $G = (V, E)$ has vertices V and directed edges E
  - $e = (u, v) \in E$ is an <u>ordered</u> pair
- By default, when we say “graph” we will mean undirected

BASIC DEFINITIONS: GRAPHS



- $V = \{1,2,3,4,5,6,7,8\}$
- $E = \{(1,2), (1,4), (1,5), (1,7), (2,3), (2,5), (3,6), (4,5)\}$
- $n = |V| = 8$
- $m = |E| = 8$

BASIC DEFINITIONS: GRAPHS



- $V = \{1,2,3,4,5,6,7,8\}$
- $E = \{(2,1), (4,1), (5,1), (1,7), (3,2), (2,5), (3,6), (4,5), (6,3)\}$
- $n = |V| = 8$
- $m = |E| = 9$

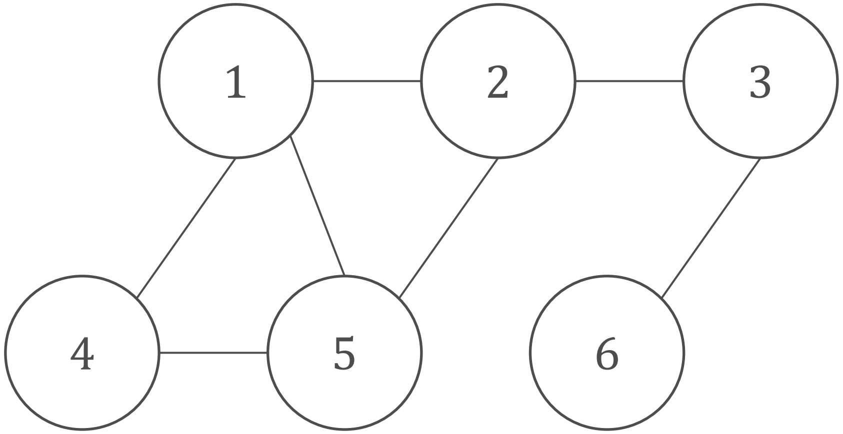
APPLICATIONS

Graph	Nodes	Edges
Transportation	Street Intersections	Highways
Communication	Computers	Optic cables
World Wide Web	Web pages	Hyperlinks
Social	People	Relationships
Software	Functions	Function calls
Scheduling	Tasks	Precedence constraints
Circuits	Gates	Wires

GRAPH REPRESENTATION: ADJACENCY MATRIX

- Adjacency matrix: an n-by-n matrix with $A_{(u,v)} = 1$ if $(u, v) \in E$

	1	2	3	4	5	6
1	0	1	0	1	1	0
2	1	0	1	0	1	0
3	0	1	0	0	0	1
4	1	0	0	0	1	0
5	1	1	0	1	0	0
6	0	0	1	0	0	0

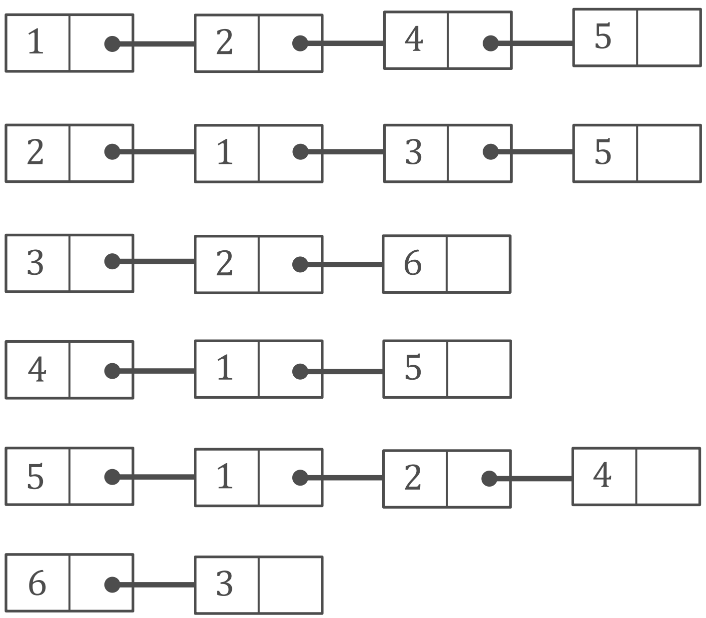


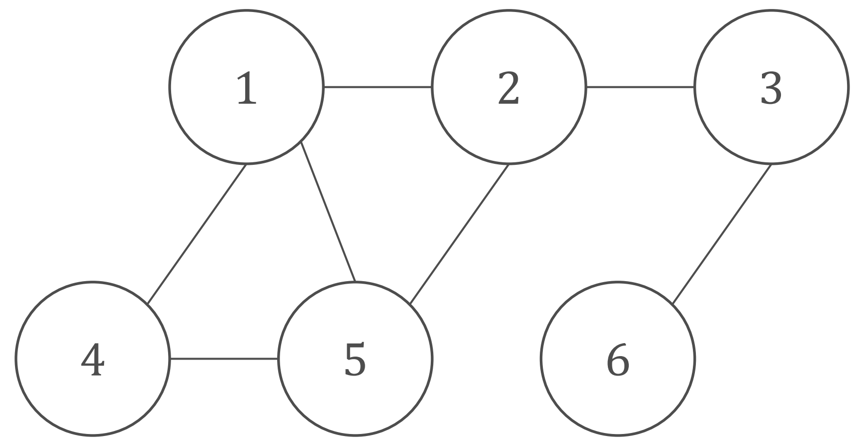
GRAPH REPRESENTATION: ADJACENCY MATRIX

- Adjacency matrix: an n-by-n matrix with $A_{(u,v)} = 1$ if $(u, v) \in E$
  - Space proportional to n^2
  - Checking if $(u, v) \in E$ takes $\Theta(1)$ time
  - Identifying all edges takes $\Theta(n^2)$ time

GRAPH REPRESENTATION: ADJACENCY LIST

- Adjacency list: Node indexed array of lists





GRAPH REPRESENTATION: ADJACENCY LIST

- Adjacency list: Node indexed array of lists
  - Two representations for each edge
  - Space proportional to $n + m$
  - Checking if $(u, v) \in E$ takes $O(\deg(u))$ time
  - Identifying all edges takes $\Theta(m + n)$ time

PATHS, CYCLES AND CONNECTIVITY

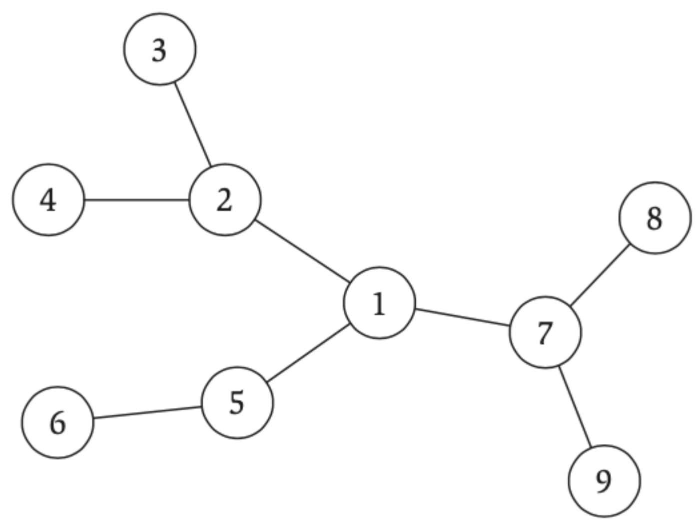
- *Definition:* A path in an undirected graph is a sequence of P of nodes $v_1, v_2, \ldots, v_k$ with the property that every consecutive pair v_i, v_{i+1} is connected by an edge
- A path is simple if all nodes are distinct
  - $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_1 \rightarrow v_5$ is not simple
- *Definition:* An undirected graph is connected if for every pair of nodes u and v there is a path between them
- *Definition:* A cycle is a path $v_1, v_2, \ldots, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k - 1$ nodes are all distinct

TREES

- ***Definition:*** An undirected graph is a tree if it is connected and does not contain a cycle.
- ***Theorem:*** Given an undirected G, any two of the statements imply the third:
  - G is connected.
  - G does not contain a cycle.
  - G has $n - 1$ edges.

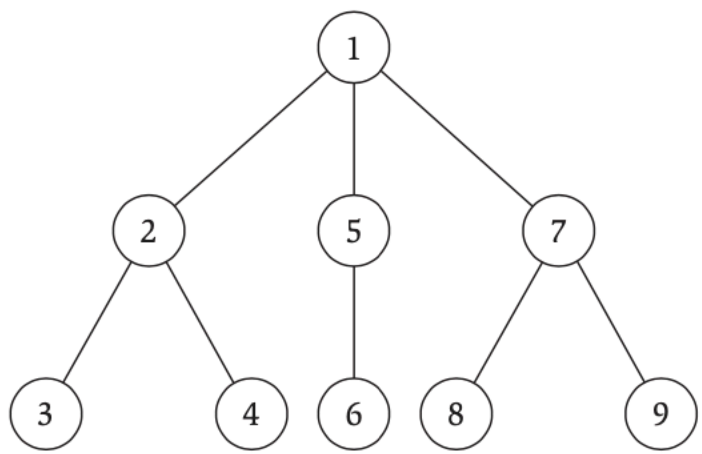
ROOTED TREE

- Given a tree T, we can choose a node r and orient each edge away from r
- Importance: model hierarchical structure



ROOTED TREE

- Given a tree T, we can choose a node r and orient each edge away from r
- Importance: model hierarchical structure

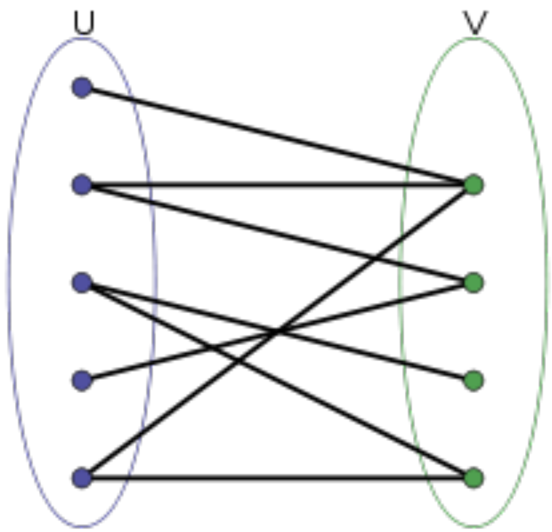


TREES

- *Definition:* A binary tree is a rooted tree where each node has at most 2 children
- *Definition:* The height of a tree is the number of edges from the longest path from root to a leaf.

BIPARTITE GRAPHS

- ***Definition:*** A graph is bipartite if the node set can be partitioned into two sets X and Y such that there is no edge (u, v) where $u, v \in X$ or $u, v \in Y$.
  - Equivalently, a graph with no odd cycles



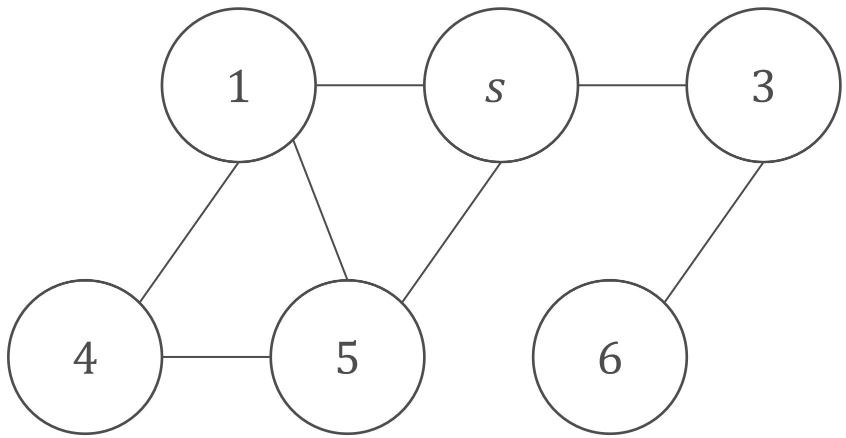
GRAPH CONNECTIVITY AND TRAVERSAL

- $s - t$ connectivity problem (today)
  - Given two nodes s, t is there a path from s to t?
- $s - t$ shortest path problem (next time)
  - Given two nodes s, t what is the length of the shortest path from s to t?
- Applications
  - Navigation
  - Transportation
  - VLSI design
  - Six degrees of separation

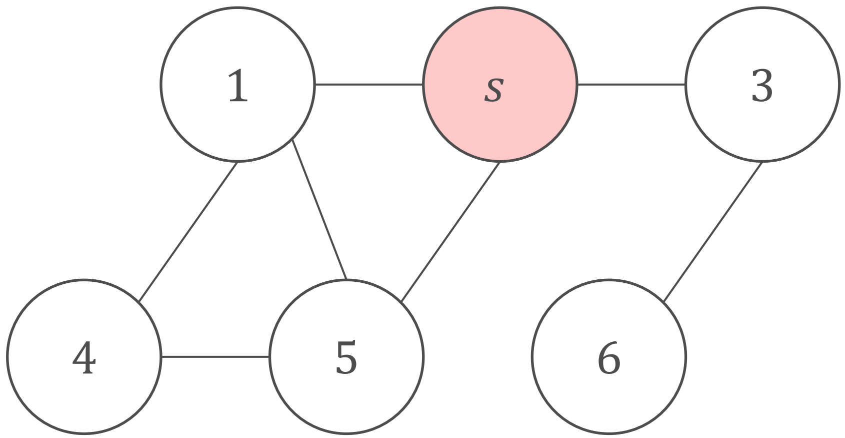
BFS: BREADTH-FIRST SEARCH

- Probably the simplest graph traversal algorithm
- Intuition: Explore ``outward'' from s in all possible directions, adding one layer at a time
  - $L_0 = \{s\}$
  - $L_1 =$ all neighbors of L_0
  - $L_2 =$ all neighbors of L_1
  - $L_{i+1} =$ all neighbors of L_i

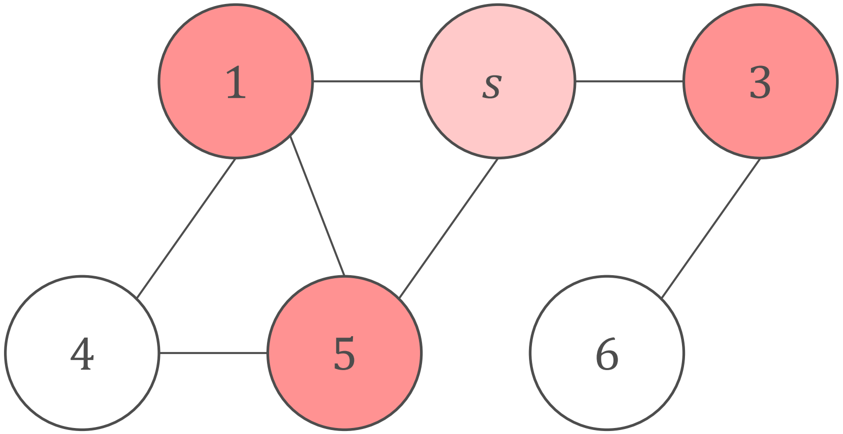
BFS



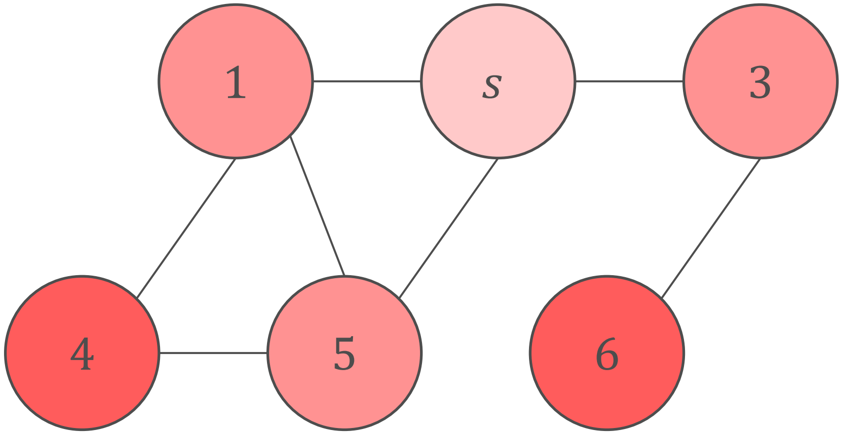
BFS



BFS



BFS

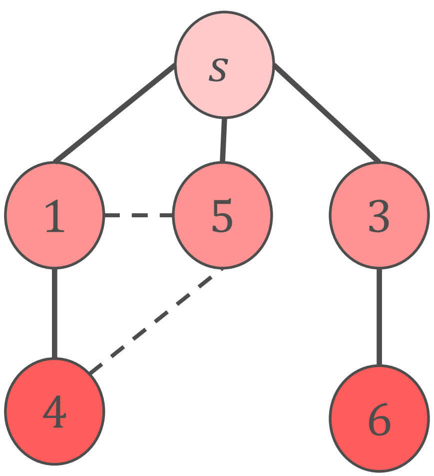


BFS

- <u>*Theorem*</u>: For each i, L_i consists of the nodes of distance exactly i from s. There is a path from s to t if and only if (iff) t appears in some layer.

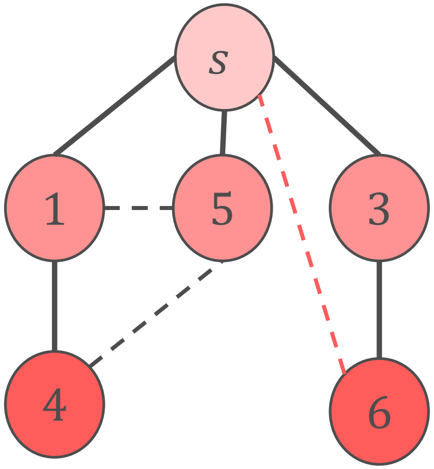
BFS

- BFS naturally defines a breadth-first tree rooted at s
- ***Property:*** Let T be a BFS tree of G and let (x, y) be an edge of G. Then the level of x and y differ by at most 1.



BFS

- BFS naturally defines a breadth-first tree rooted at s
- ***Property:*** Let T be a BFS tree of G and let (x, y) be an edge of G. Then the level of x and y differ by at most 1.



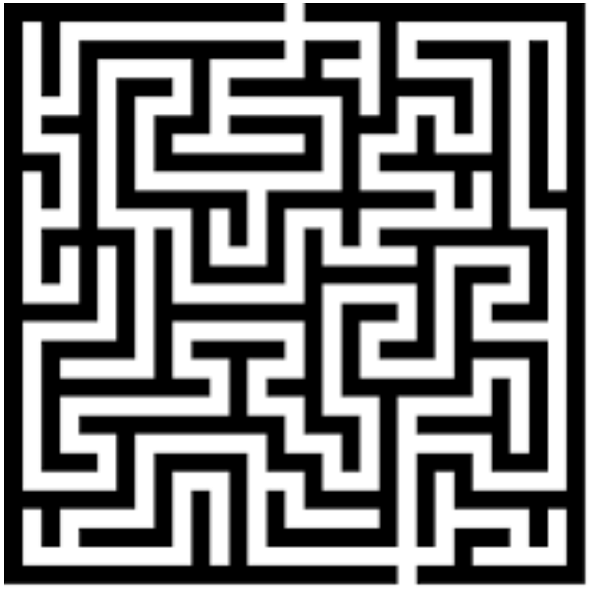
Proof: Suppose there existed (u, v) such that $u \in L_i$ and $v \in L_{i+2}$. Since $u \in L_i$, by definition of L_{i+1} it should contain all neighbors of u; contradiction.

BFS

- Theorem: BFS runs in $O(m+n)$ time if the graph is given by its adjacency representation.
- Easy to see $O(n^2)$
  - At most n lists
  - Each node occurs at most once in each list
  - When we consider a vertex u, there are at most $\deg(u) \leq n$ neighbors/edges to process
- Slightly better analysis to get $O(m+n)$
  - When we consider a vertex u there are at most $\deg(u)$ neighbors
  - Total time processing edges $\sum_{u \in V} \deg(u) = 2m$

DFS: DEPTH FIRST SEARCH

- Maze exploration



DFS: DEPTH FIRST SEARCH

- $DFS(v)$:
  - If v is unmarked:
    - Mark v;
    - For each edge (v, u):
      - $DFS(u)$
- ***Claim:*** $DFS(v)$ runs in time $O(m + n)$
- ***Claim:*** $DFS(v)$ visits all nodes reachable from v

DFS: DEPTH FIRST SEARCH

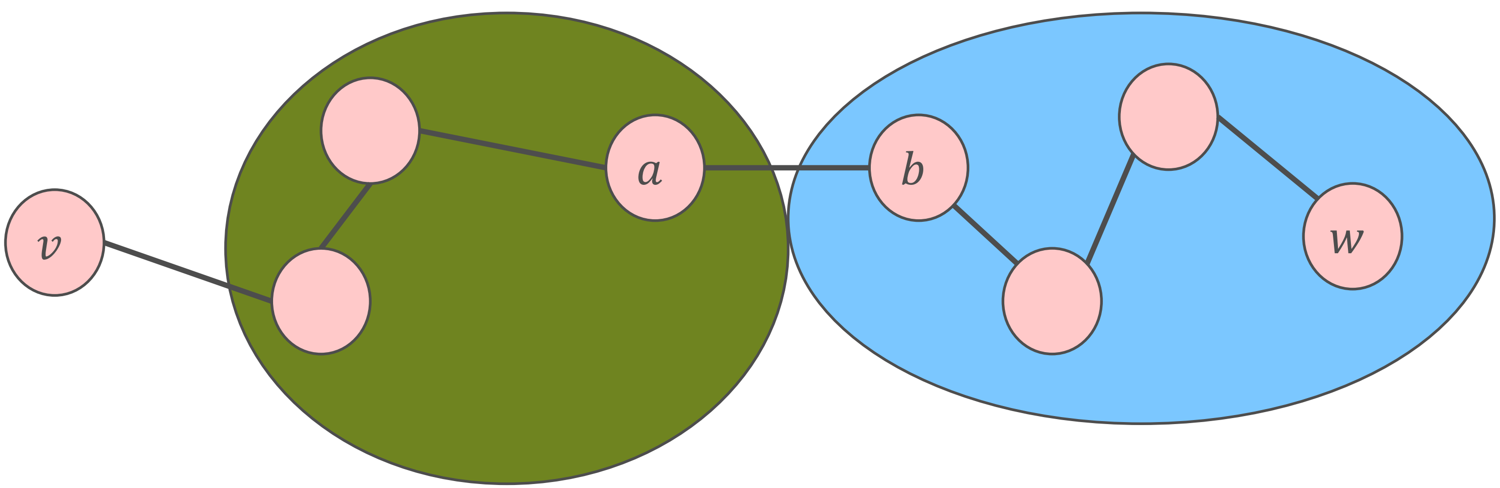
- *Claim:* $DFS(v)$ runs in time $O(m+n)$
- Proof:
  - We traverse an edge (u, v) only after we have marked u
  - Each node u gets marked once
  - Therefore, each edge is traversed at most once

DFS: DEPTH FIRST SEARCH

- *Claim:* $DFS(v)$ visits all nodes reachable from v
- Proof:
  - Let A be the set of marked vertices at the end
  - Let $B = V \setminus A$ be the unmarked vertices
  - Let w be a vertex reachable from v, but $w \in B$

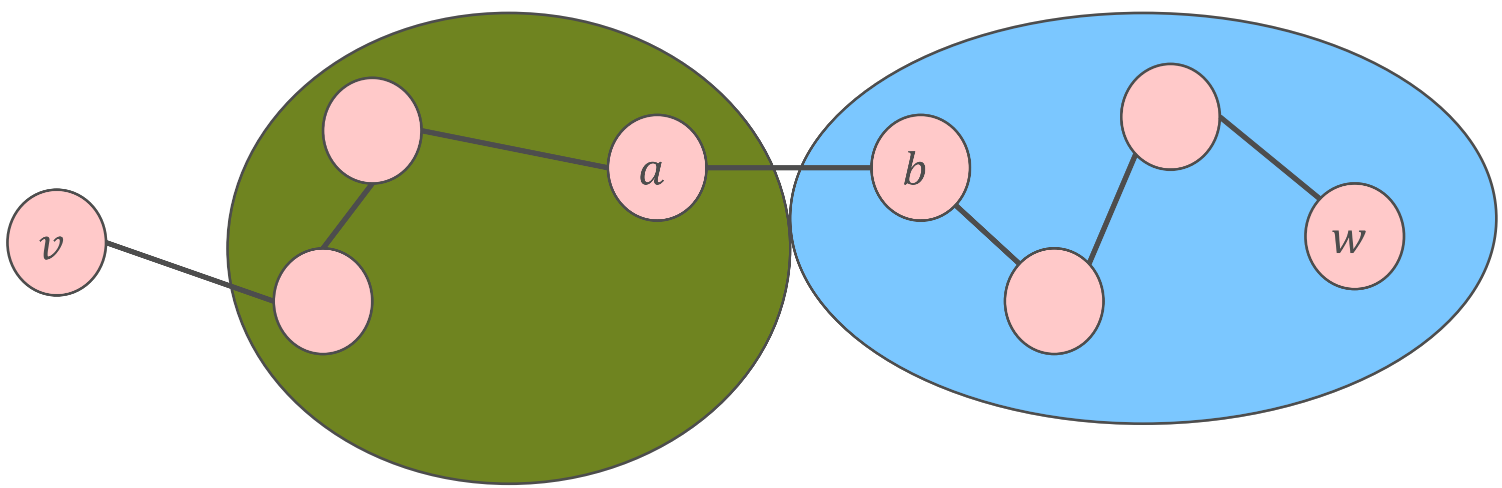
DFS: DEPTH FIRST SEARCH

- *Claim:* $DFS(v)$ visits all nodes reachable from v
- Proof:
  - Let A be the set of marked vertices at the end
  - Let $B = V \setminus A$ be the unmarked vertices
  - Let w be a vertex reachable from v, but $w \in B$



DFS: DEPTH FIRST SEARCH

- *Claim:* $DFS(v)$ visits all nodes reachable from v
- Proof cont.:
  - There must be an edge (a, b) in the path from v to w that we didn't traverse, such that $a \in A$ and $b \in B$



DIRECTED GRAPHS

CONNECTIVITY IN DIRECTED GRAPHS

- *Directed reachability*: Given a node s, find all nodes reachable from s
  - BFS and DFS naturally extend to directed graphs
  - Done!

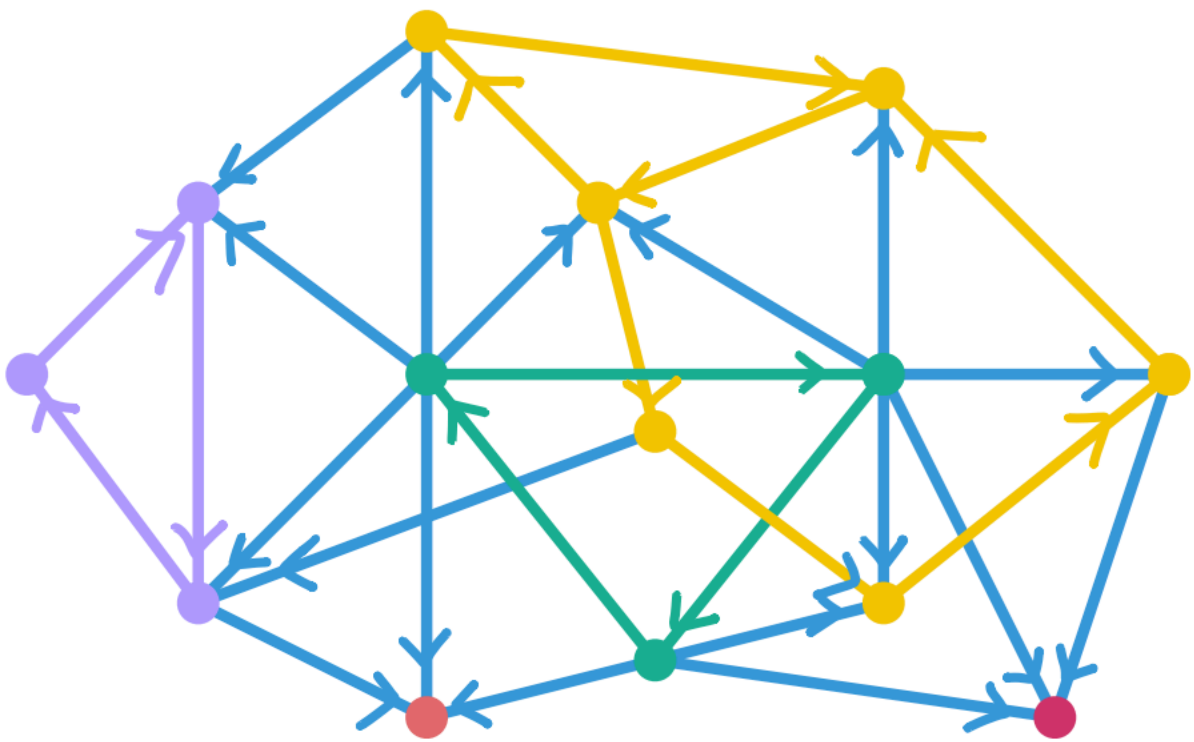
STRONG CONNECTIVITY

- Nodes u and v are mutually reachable if there is a path from u to v and from v to u
- A graph is strongly connected if every pair of nodes is mutually reachable
- Strong component of v: the set of vertices s such that v and s are mutually reachable
  - Claim: For any two nodes u and v, their strong components are either identical or disjoint

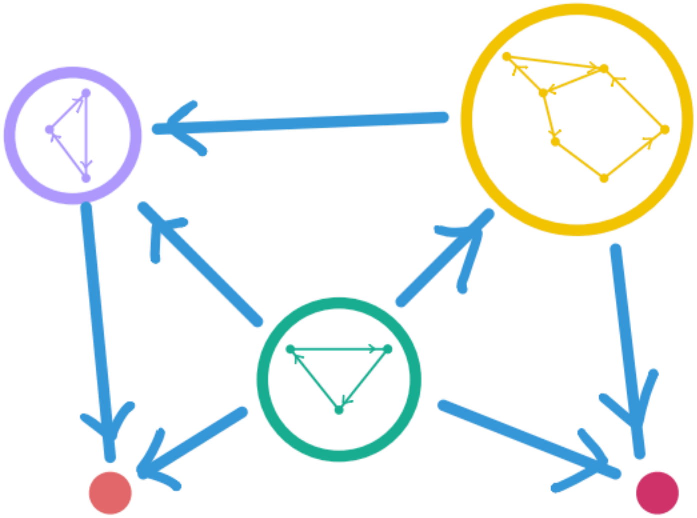
STRONG CONNECTIVITY

- How do we find the strong component of v?
  - Run DFS from v
  - Run DFS from v in the reversed graph
    - Finds vertices that can reach v in original graph
  - Return all vertices visited in <u>both</u> executions
- Repeat for all v to find all strongly connected components

STRONG CONNECTIVITY

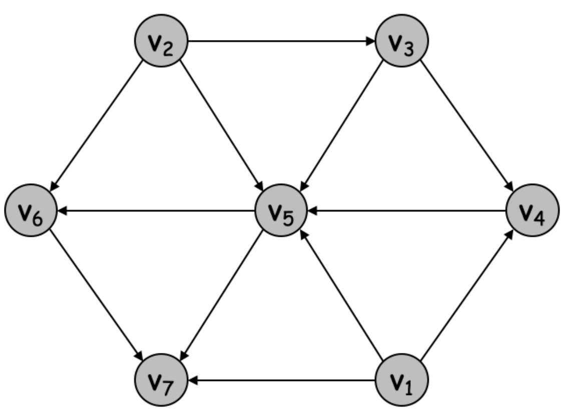


STRONG CONNECTIVITY

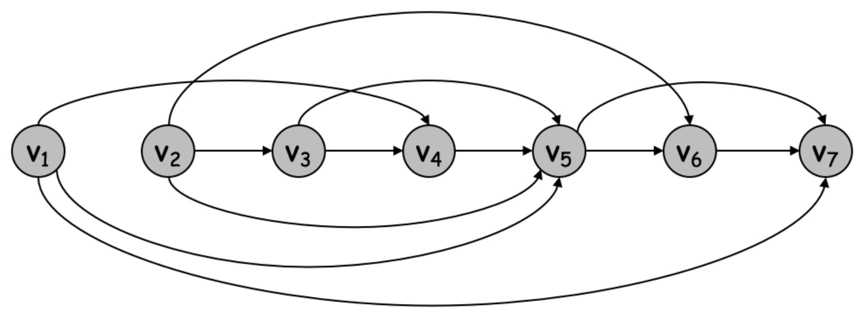


DAGS AND TOPOLOGICAL ORDERING

- A DAG (Directed Acyclic Graph) is a directed graph that contains no cycles
- A topological order of a directed graph is an ordering of its nodes $v_1, v_2, \ldots, v_n$ such that for every edge (v_i, v_j) we have that $i < j$



a DAG



a topological ordering

DAGS AND TOPOLOGICAL ORDERING

- Lemma: If G has a topological ordering then it is a DAG
- Proof:
  - Let $v_1, \ldots, v_n$ be the topological ordering, and assume there is a cycle C
  - Let v_i be the lowest index vertex in the cycle, and v_j be the vertex that points to v_i in C, i.e. C contains the edge (v_j, v_i)
  - By the choice of i we have $i < j$, and therefore there is a backwards edge in the topological order; a contradiction

DAGS AND TOPOLOGICAL ORDERING

- Lemma 1: If G is a DAG it has a topological order
- Proof:
  - Claim 1: If G is a DAG, there exists a node v with no incoming edges
  - Proving Claim 1 proves the lemma:
    - Find the v guaranteed by Claim 1 and put it first in the topological order
    - $G' = G - \{v\}$ is a DAG: deleting cannot create cycles
    - Recurse on G'

DAGS AND TOPOLOGICAL ORDERING

- Claim 1: If G is a DAG, there exists a node v with no incoming edges
- Proof:
  - Suppose that G is a DAG and for every node u there exists a node w such that $(w, u) \in E$
  - Pick any node u and walk "backwards"
  - This process can continue forever, since every node has at least once incoming edge
  - Since there are at most n vertices in the first $n + 1$ steps of this process some vertex w appears twice
  - We have found a cycle! A contradiction.

DAGS AND TOPOLOGICAL ORDERING

- Computing a topological order of a DAG
- The proof of Lemma 1 naturally extends to an algorithm:
  - Find a node v with no incoming edges
  - Put v first in the topological order
  - Delete v from G
  - Compute a topological order of $G - \{v\}$ and append after v

SUMMARY

- A lot definitions!
- Reachability
- Basic graph algorithms: BFS and DFS
- SCCs (strongly connected components)
- DAGS and topological ordering

SUMMARY

- Only the beginning…
- Suppose we only have $O(\log(n))$ space
  - Just enough space to write down the id of a constant number of nodes
- Question: Can we solve reachability?
- Undirected graphs, deterministically:
  - Yes! Reingold (2004)
- Directed graphs, deterministically:
  - Open!