CSE 331: Algorithms & Complexity

“Graph Theory”

Prof. Charlie Anne Carlson (She/Her)

Lecture 10

Friday September 19th, 2025

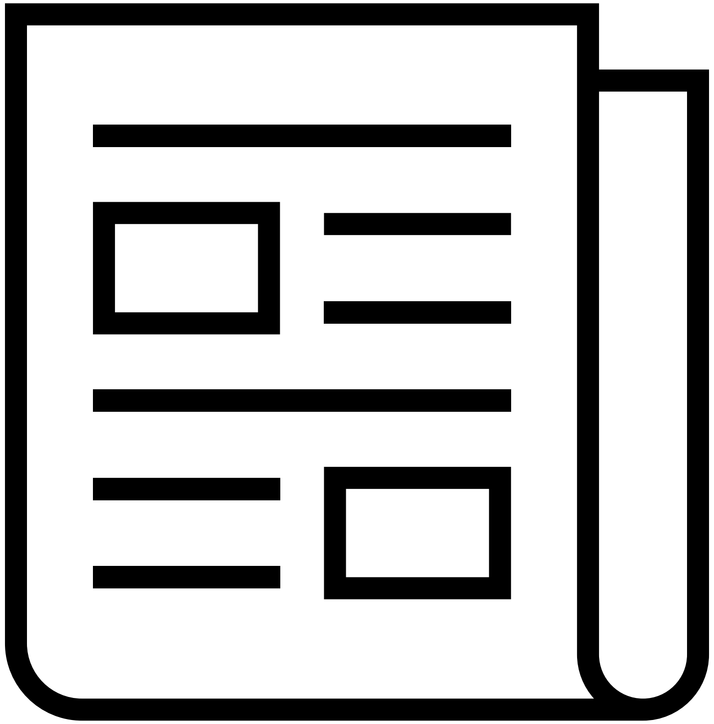
University at Buffalo

Schedule

1. Course Updates
2. Finish GS Analysis
3. Graphs
    1. Connectivity
    2. Trees
    3. Traversal

Course Updates

- HW 1 Solutions Out
- HW 2 Out
- Project Signup end of day
- First Quiz Monday Sept 29
  - In class
  - Check Piazza for practice problems



Student's Current State

GALE–SHAPLEY (*preference lists for n hospitals and n students*)

INITIALIZE *M* to empty matching.

WHILE (some hospital *h* is unmatched)

  s ← first student on *h*'s list to whom *h* has not yet proposed.

  IF (*s* is unmatched)

    Add *h*–*s* to matching *M*.

  ELSE IF (*s* prefers *h* to current partner *h'*)

These are not the indices of hospitals or students but preferences.

RETURN stable matching *M*.

- `H[i,j]` is hospital i's jth preferered student.
- `S[i,j]` is student i's jth preferred hospital.
- Unmatched Hospitals stored in a linked list called `H_unmatched`.
- `Next[i]` is hospital i's "next preference" to ask.
- `Current[i]` is student i's current "matched preference."

Student's Current State

GALE–SHAPLEY (*preference lists for n hospitals and n students*)

INITIALIZE M to empty matching.

WHILE (some hospital h is unmatched)

$s \leftarrow$ first student on h's list to whom h has not yet proposed.

IF (s is unmatched)

Add h–s to matching M.

ELSE IF (s prefers h to current partner h')

Replace h'–s with h–s in matching M.

ELSE

s rejects h.

RETURN stable matching M.

- How do we do this check?

Student's Current State

GALE–SHAPLEY (*preference lists for n hospitals and n students*)

INITIALIZE M to empty matching.

WHILE (some hospital h is unmatched)

  s ← first student on h's list to whom h has not yet proposed.

  IF (s is unmatched)

    Add h–s to matching M.

  ELSE IF (s prefers h to current partner h')

    Replace h'–s with h–s in matching M.

  ELSE

    s rejects h.

RETURN stable matching M.

- Create an array of arrays, `Ranking` such that for student i and hospital j, `Ranking[i,j]` is the rank of j for i.
  - Not the same as `S` but similar.
- We can can construct this for all students in O(n^2) time before the loop.

Student's Current State

GALE–SHAPLEY (*preference lists for n hospitals and n students*)

INITIALIZE M to empty matching.

WHILE (some hospital h is unmatched)

  $s \leftarrow$ first student on h's list to whom h has not yet proposed.

  IF (s is unmatched)

    Add h–s to matching M.

  ELSE IF (s prefers h to current partner h')

    Replace h'–s with h–s in matching M.

  ELSE

    s rejects h.

RETURN stable matching M.

- Create an array of arrays, `Ranking` such that for student i and hospital j, `Ranking[i,j]` is the rank of j for i.
- We can then in O(1) time compare `Ranking[s,h]` and `Ranking[s,Current[s]]`.

Returning Matching

GALE–SHAPLEY (*preference lists for n hospitals and n students*)

INITIALIZE M to empty matching.

WHILE (some hospital h is unmatched)

$s \leftarrow$ first student on h's list to whom h has not yet proposed.

IF (s is unmatched)

Add h–s to matching M.

ELSE IF (s prefers h to current partner h')

Replace h'–s with h–s in matching M.

ELSE

s rejects h.

RETURN stable matching M.

- Where is our matching?

Returning Matching

GALE–SHAPLEY (*preference lists for n hospitals and n students*)

INITIALIZE M to empty matching.

WHILE (some hospital h is unmatched)

$s \leftarrow$ first student on h's list to whom h has not yet proposed.

IF (s is unmatched)

Add h–s to matching M.

ELSE IF (s prefers h to current partner h')

Replace h'–s with h–s in matching M.

ELSE

s rejects h.

RETURN stable matching M.

- The matching is stored in `Current`
- You can convert this into a list of pairs by looping over all students, finding the current match and adding the pair to `M`.
- This takes O(n) time.

A: … $\leq O(n^2) + n^2 \cdot O(1) + O(n^2) \leq O(n^2)$

GALE–SHAPLEY (*preference lists for n hospitals and n students*)

INITIALIZE M to empty matching.

WHILE (some hospital h is unmatched)

  $s \leftarrow$ first student on h's list to whom h has not yet proposed.

  IF (s is unmatched)

    Add h–s to matching M.

  ELSE IF (s prefers h to current partner h')

    Replace h'–s with h–s in matching M.

  ELSE

    s rejects h.

RETURN stable matching M.

$T_0 \in O(n^2)$ Computations

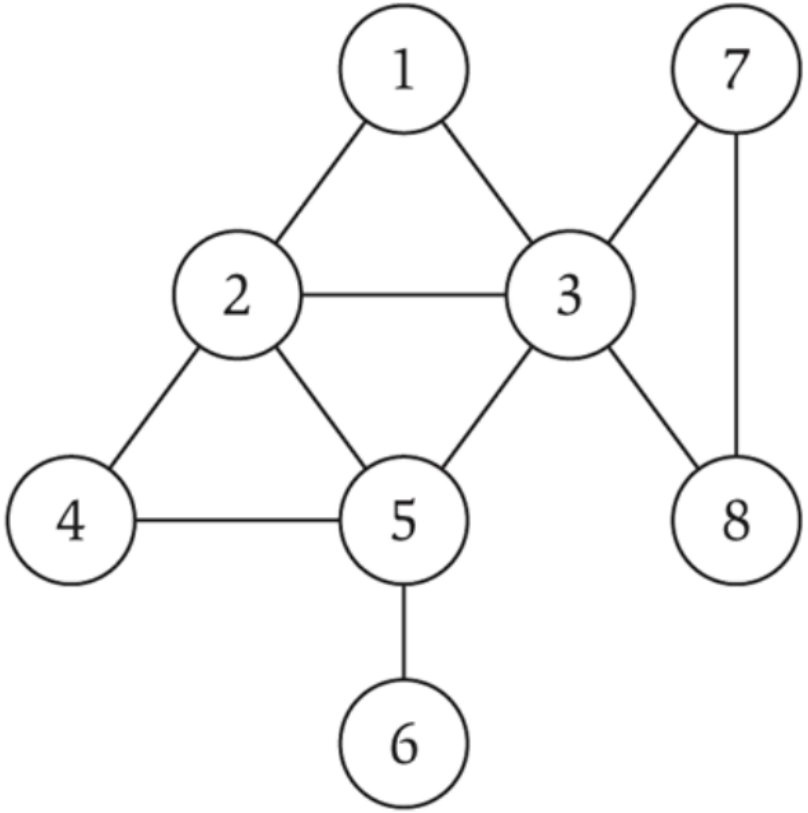
$T_1 \leq n^2$ Number Loops

$T_2 \in O(1)$ Computations

$T_3 \in O(n^2)$ Computations

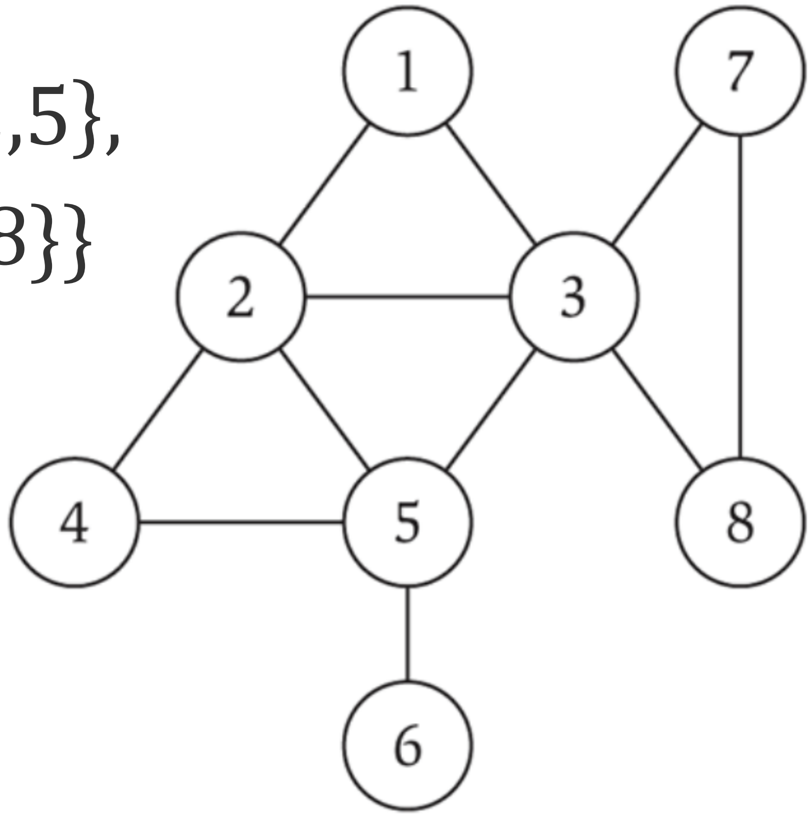
Graph G = (V,E)

- A graph is a way of encoding pairwise relationships on a set of objects.
- You have a collection V of **vertices or nodes** and
- a collection E **edges** that "connect" two nodes.
- We often define $n = |V|$ and $m = |E|$.



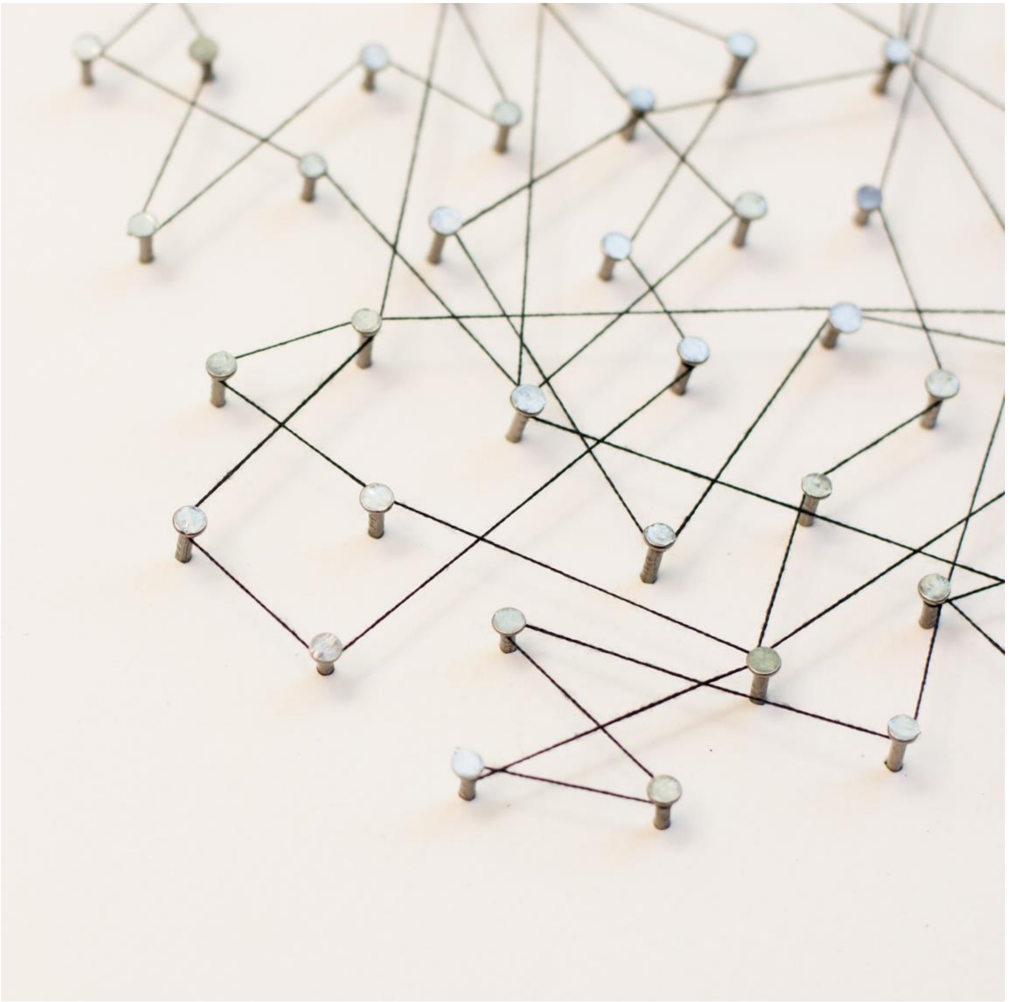
Graph G = (V,E)

- $V = \{1,2,3,4,5,6,7,8\}$
- $E = \{ \{1,2\}, \{1,3\}, \{2,3\}, \{2,4\}, \{2,5\}, \{3,5\}, \{3,7\}, \{3,8\}, \{4,5\}, \{5,6\}, \{7,8\}\}$
- $n = 8$
- $m = 11$



Examples of Graphs

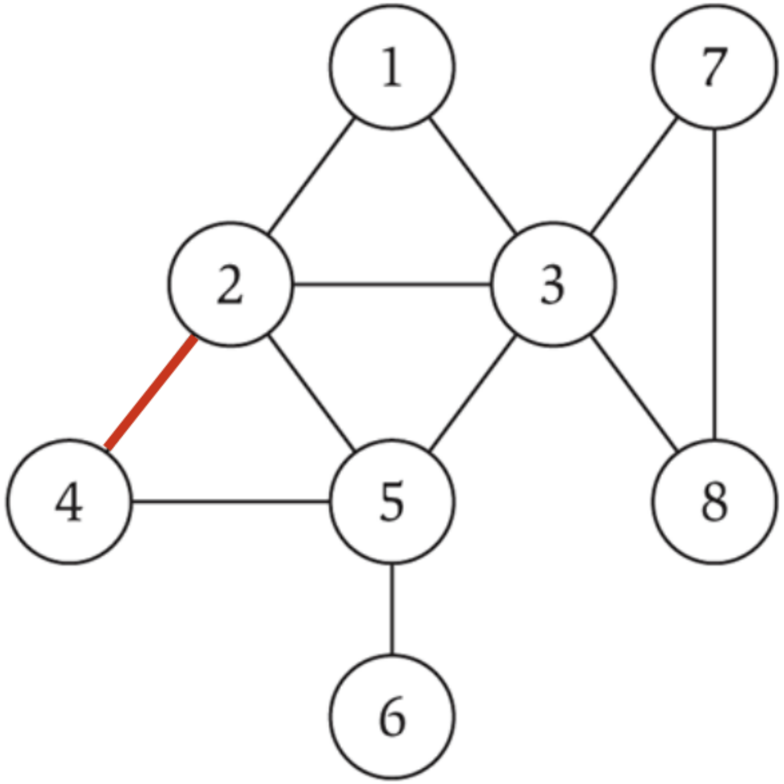
- Social Networks
- Maps
- Computer Networks
- Neural Networks
- Circuits
- Molecules Representations
- ”Everything”



Graph Representation: Adjacency Matrix

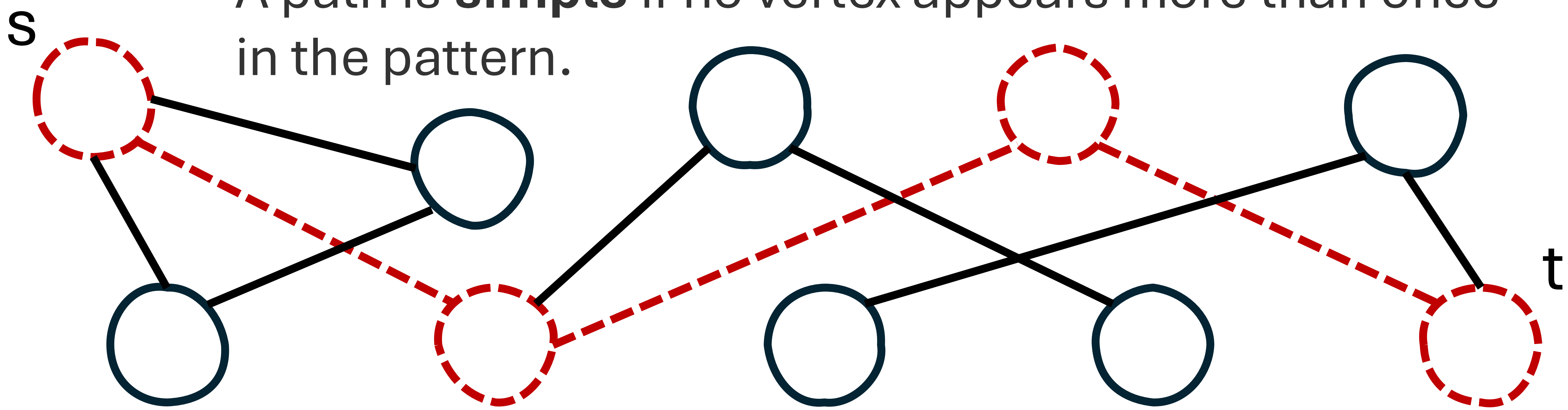
- The **adjacency matrix** is an n-by-n matrix A such that $A(i,j) = 1$ if $\{i,j\} \in E$ and 0 otherwise.

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0



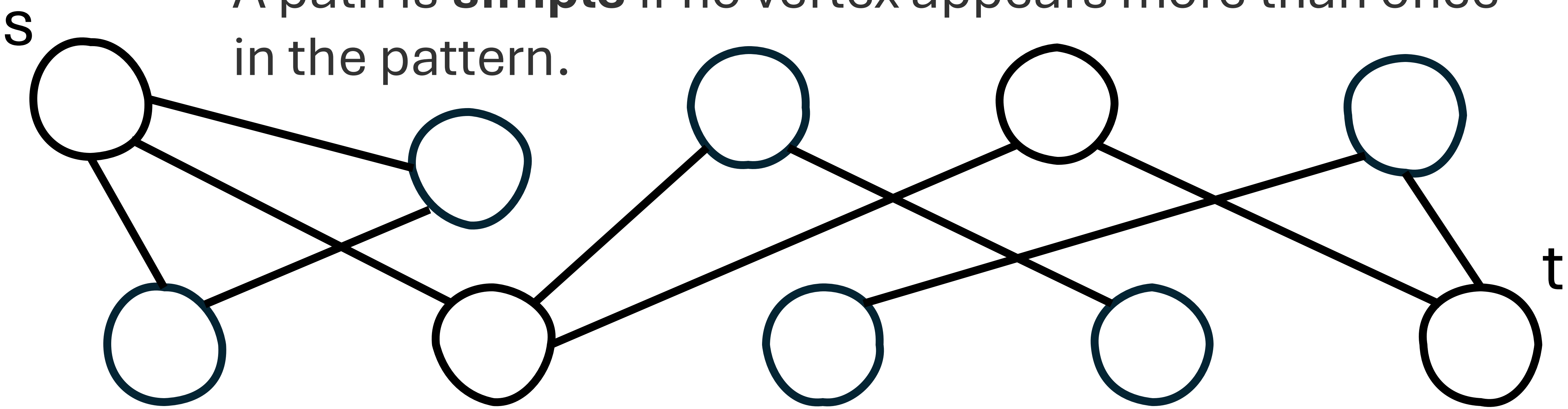
Paths and Connectivity

- A **path** in an undirected graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \dots, v_k$ such that for every consecutive pair v_i and v_{i+1}, $\{v_i, v_{i+1}\} \in E$.
  - A path is **simple** if no vertex appears more than once in the pattern.



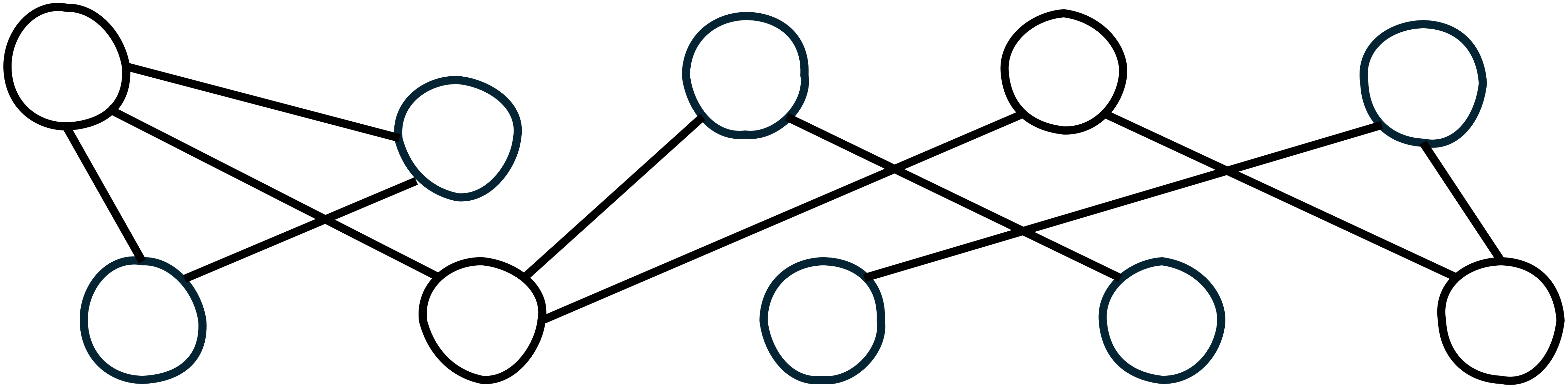
Paths and Connectivity

- A **path** in an undirected graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that for every consecutive pair v_i and v_{i+1}, $\{v_i, v_{i+1}\} \in E$.
  - A path is **simple** if no vertex appears more than once in the pattern.



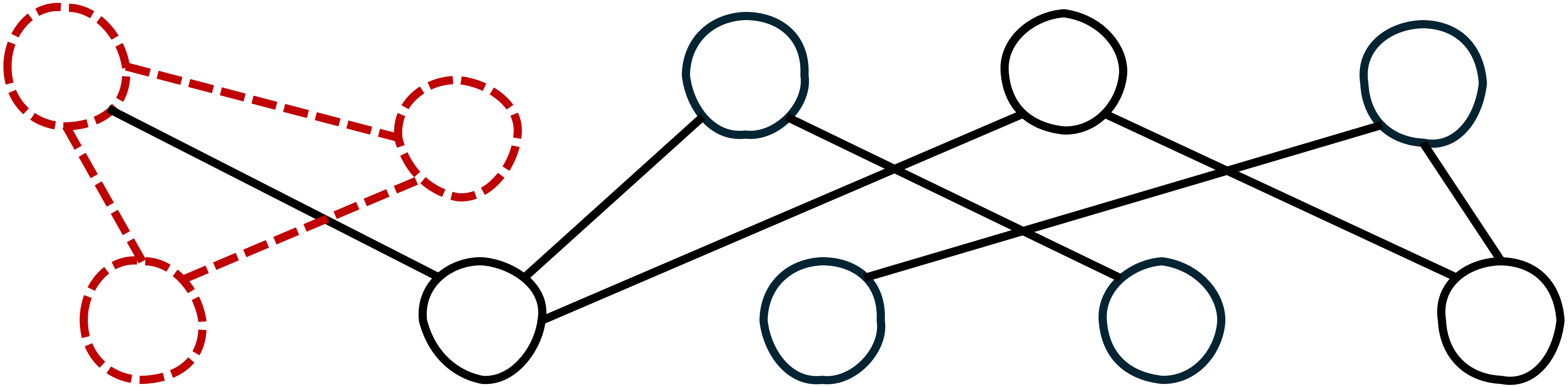
Paths and Connectivity

- A graph $G = (V, E)$ is **connected** if for any $u \in V$ and $v \in V$, there exists a simple path between u and v.



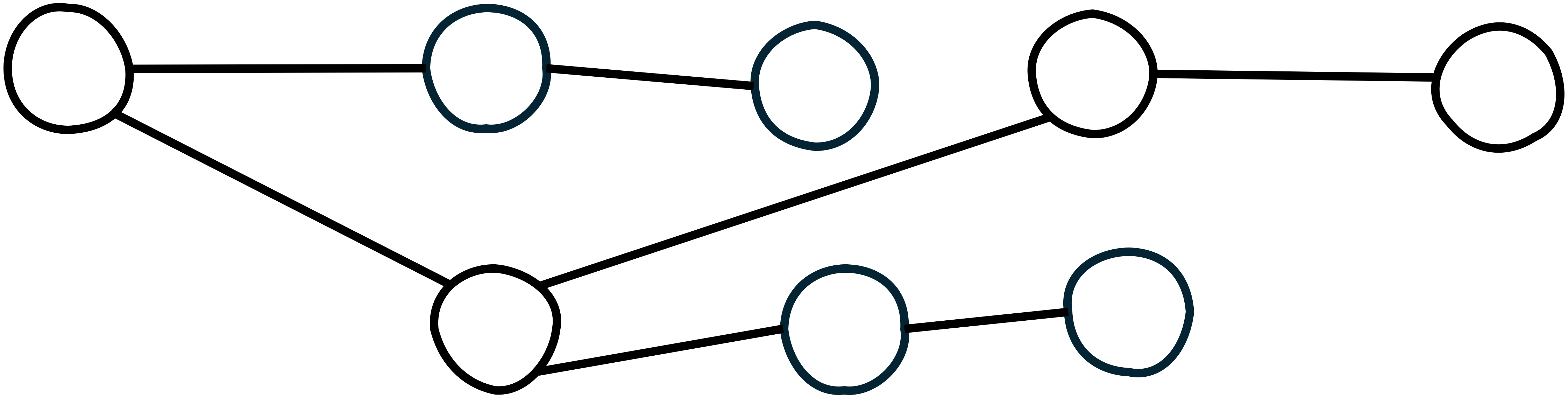
Cycles

- A **cycle** in a graph $G = (V, E)$ is a path $v_1, v_2, \ldots, v_k$ such that $k \geq 2$ and $v_1 = v_k$ (it starts and ends at the same vertex).
  - We say it is **simple** if there are no other repeats.



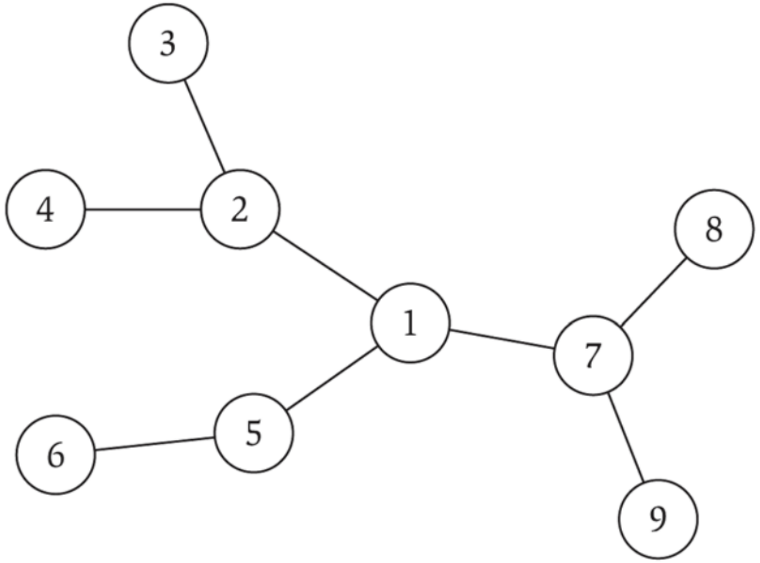
Trees

- An undirected graph is a **tree** if it is connected and does not contain a cycle.
    - A graph that doesn't contain a cycle is called **acyclic**.
    - A vertex with degree 1 is called a **leaf**.

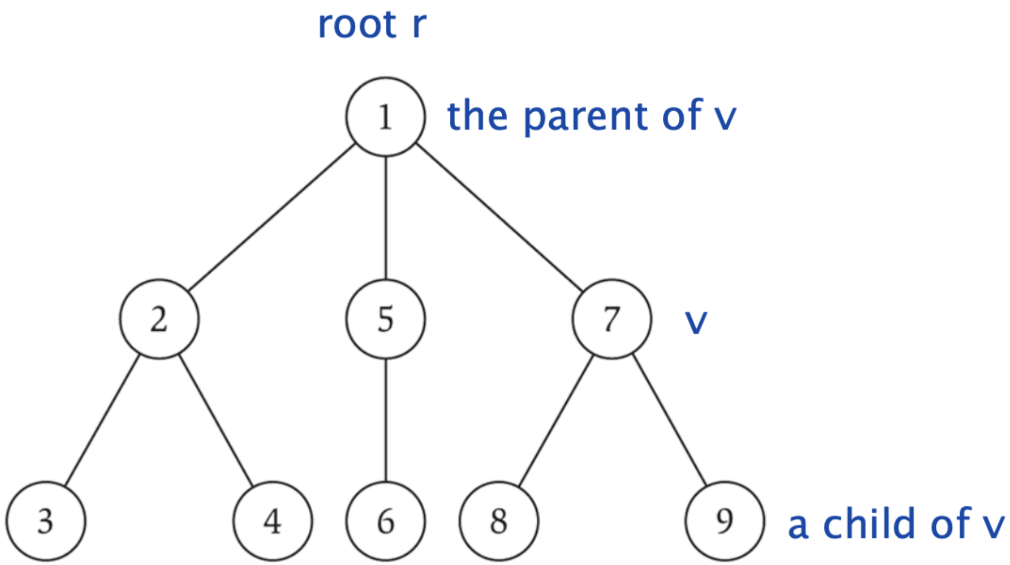


Rooted Trees

- A **rooted tree** is a tree with some node r fixed as the **root**. We think of it as a hierarchical structure.



a tree

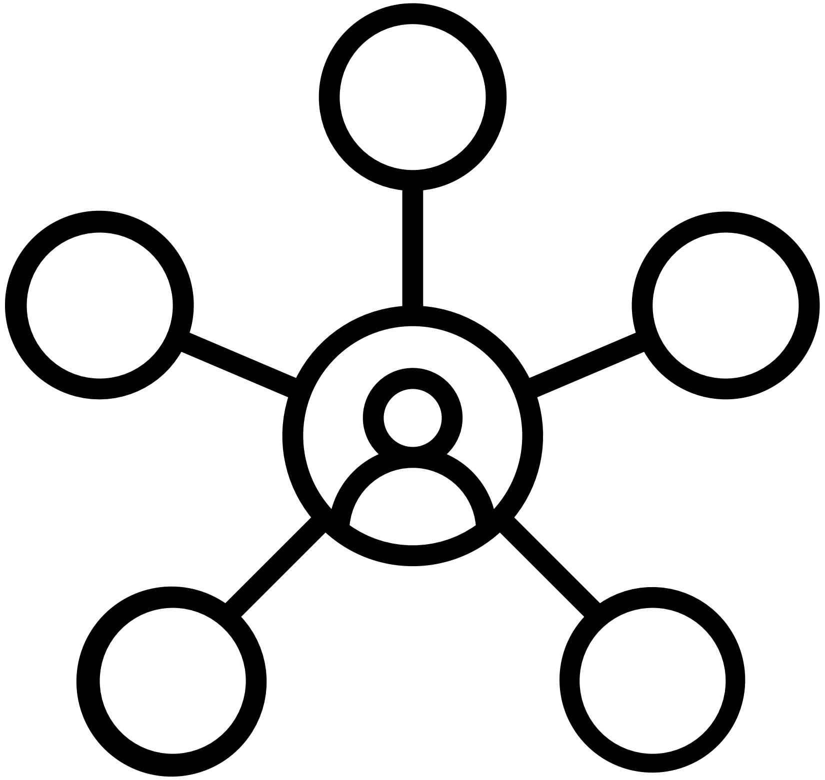


the same tree, rooted at 1

Trees

Lemma: Let $G = (V, E)$ be an undirected graph on n nodes. Any two of the following statements imply the third:

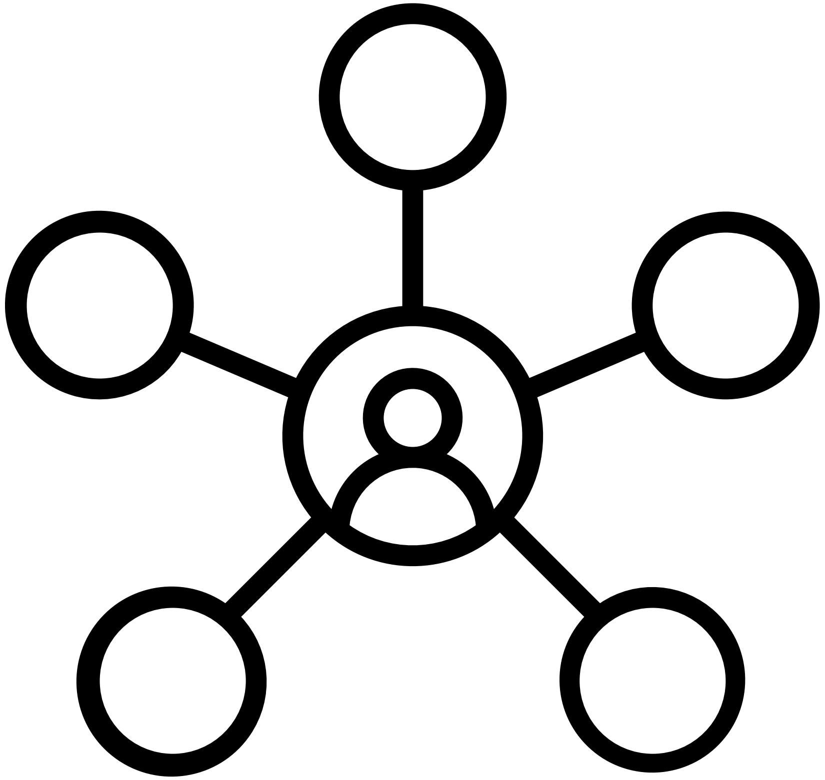
- G is connected.
- G does not contain a cycle.
- G has $n - 1$ edges.



Trees (Proof on Website)

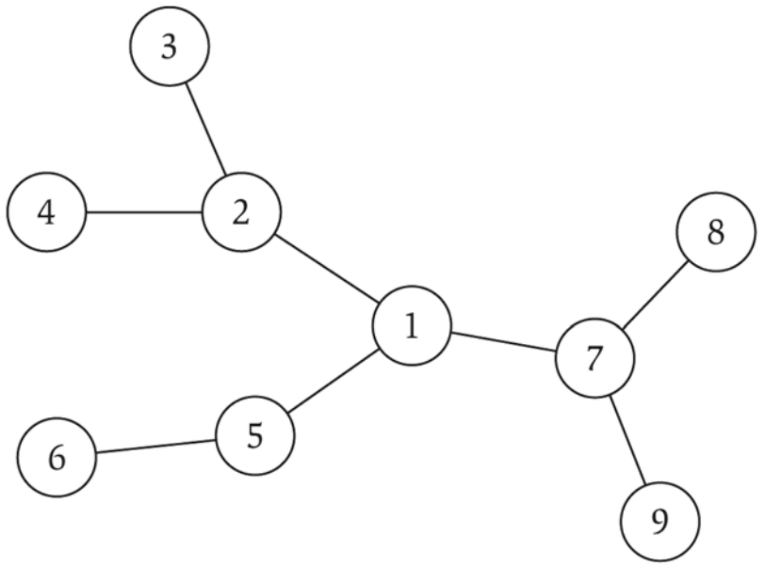
Lemma: Let $G = (V, E)$ be an undirected graph on n nodes. Any two of the following statements imply the third:

- G is connected.
- G does not contain a cycle.
- G has $n - 1$ edges.

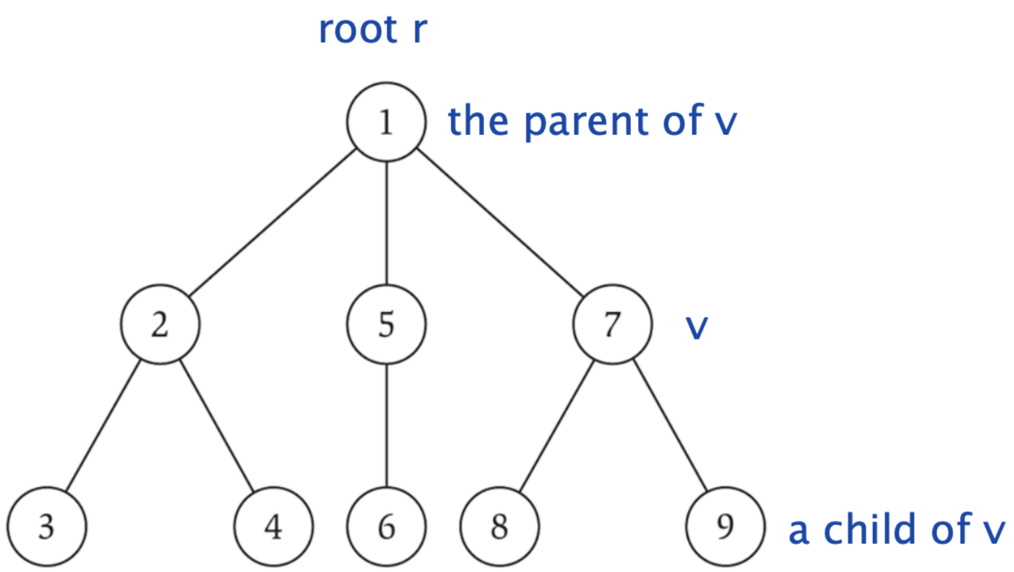


Trees

Lemma: Let $\mathrm{T} = (V, E)$ be a tree on n nodes. Then if $n \geq 2$, there must exist at least two leaf nodes.



a tree



the same tree, rooted at 1

Connectivity Problem(s)

s-t Connectivity Problem:

Input: Graph $G = (V, E)$, source s, and destination t.

Output: True if there exists a path and False otherwise.

s-t Shortest Path Problem:

Input: Graph $G = (V, E)$, source s, and destination t.

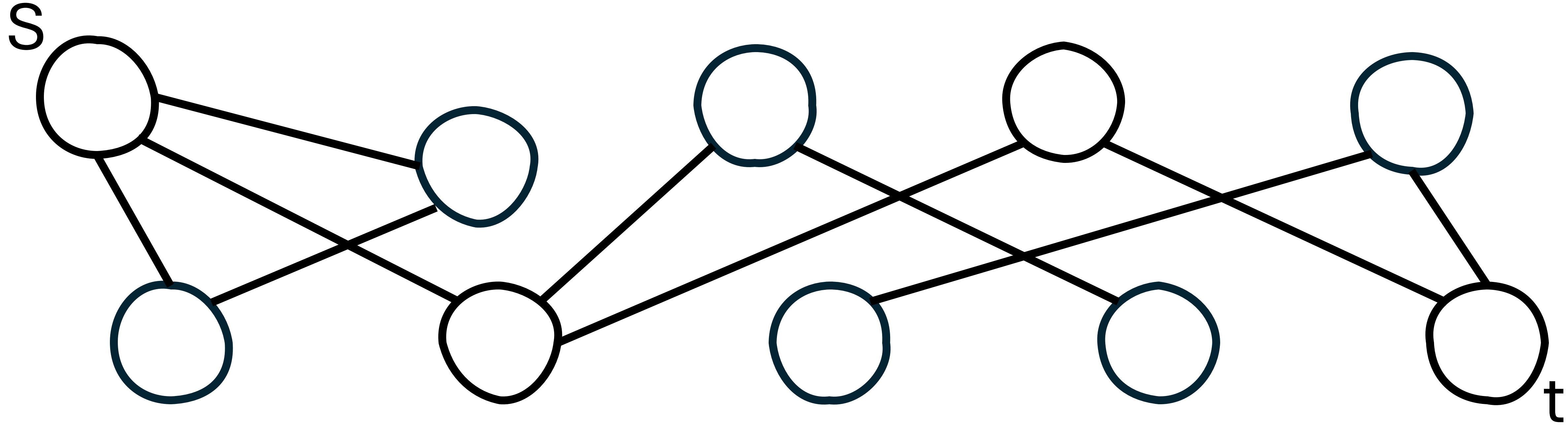
Output: The length of the shortest path from s to t (∞ if there is no path).

Q: What is a brute force solution?

s-t Connectivity Problem:

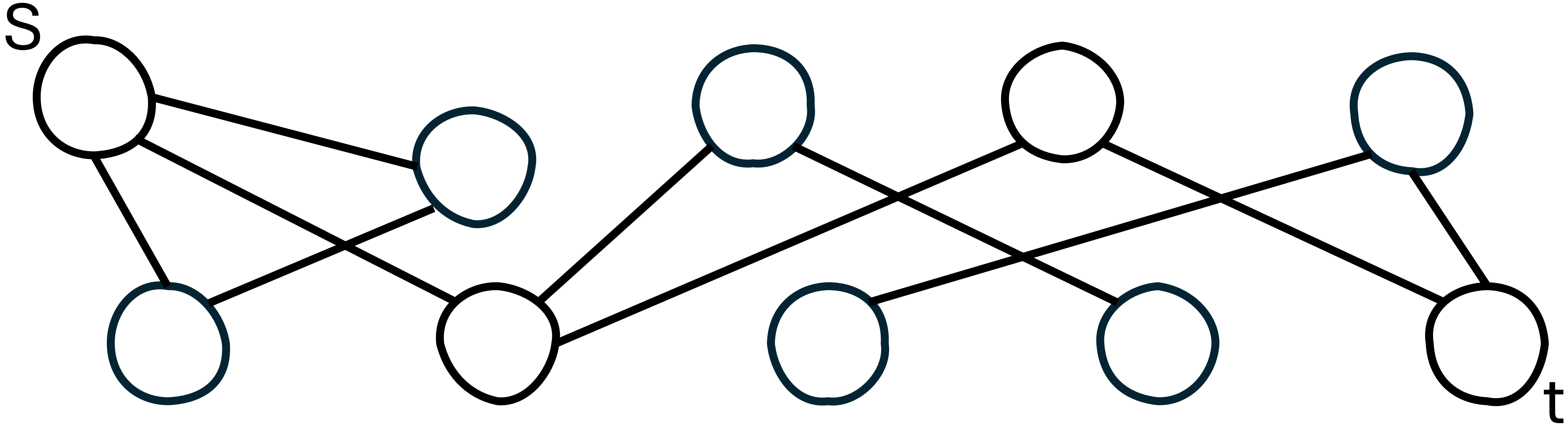
Input: Graph $G = (V, E)$, source s, and destination t.

Output: True if there exists a path and False otherwise.



Q: What is a brute force solution?

You could generate all sequences of length n-1 and check if any are paths. However, there are a lot of sequences ($(n-2)^{n-1}$). That is slow...

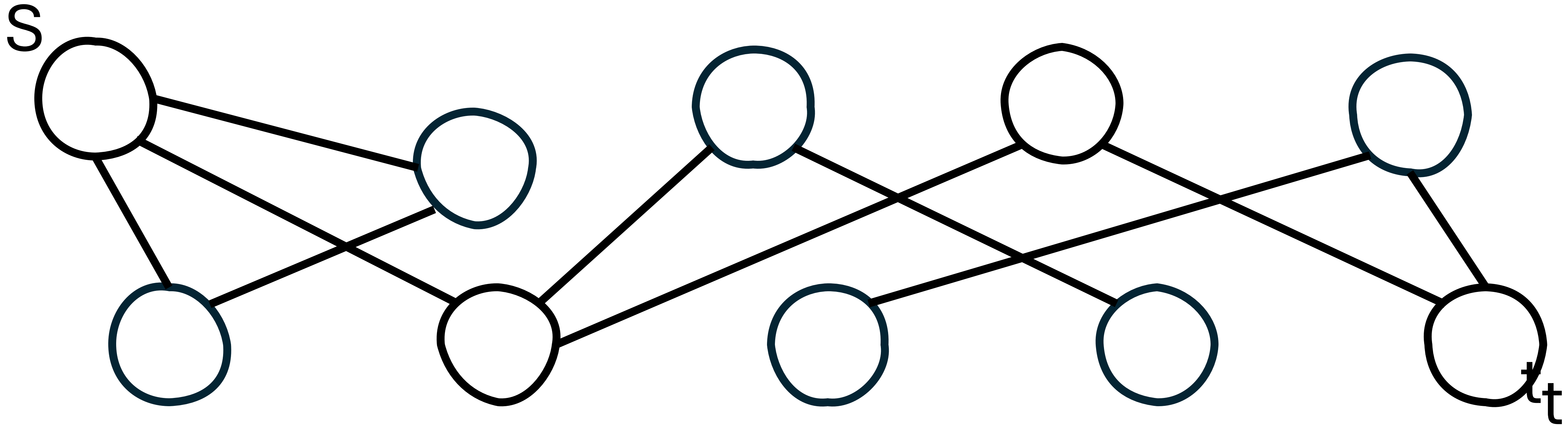


Another Connectivity Problem

Graph Connectivity:

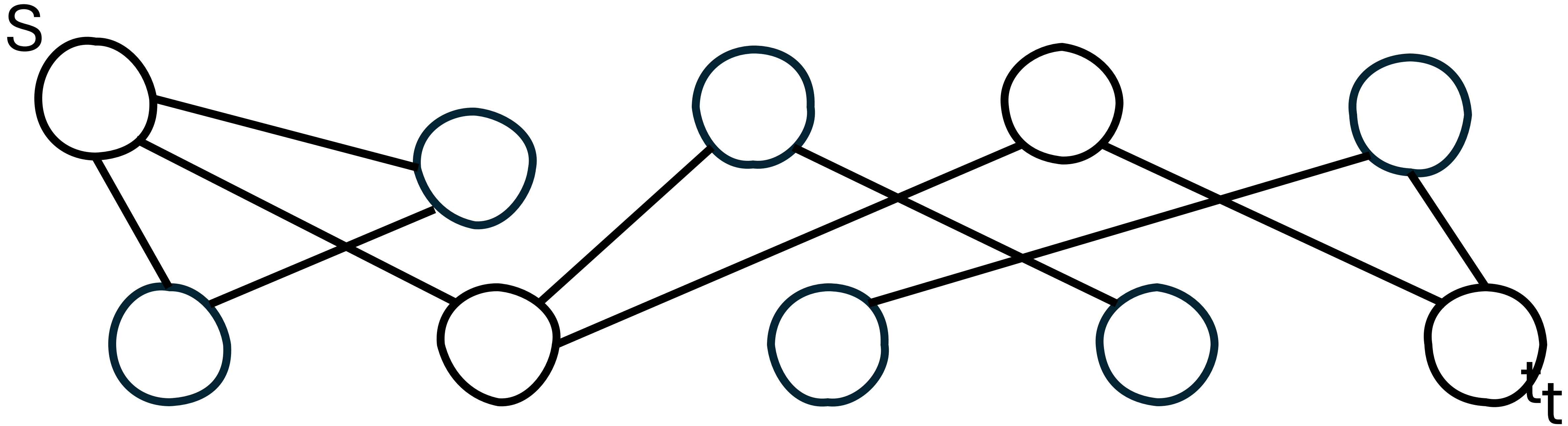
Input: $G = (V, E)$ and $s \in V$

Output: $\mathrm{T} = \{\mathrm{u} \in \mathrm{T}: \text{ there exists a path from s to u in G}\}$



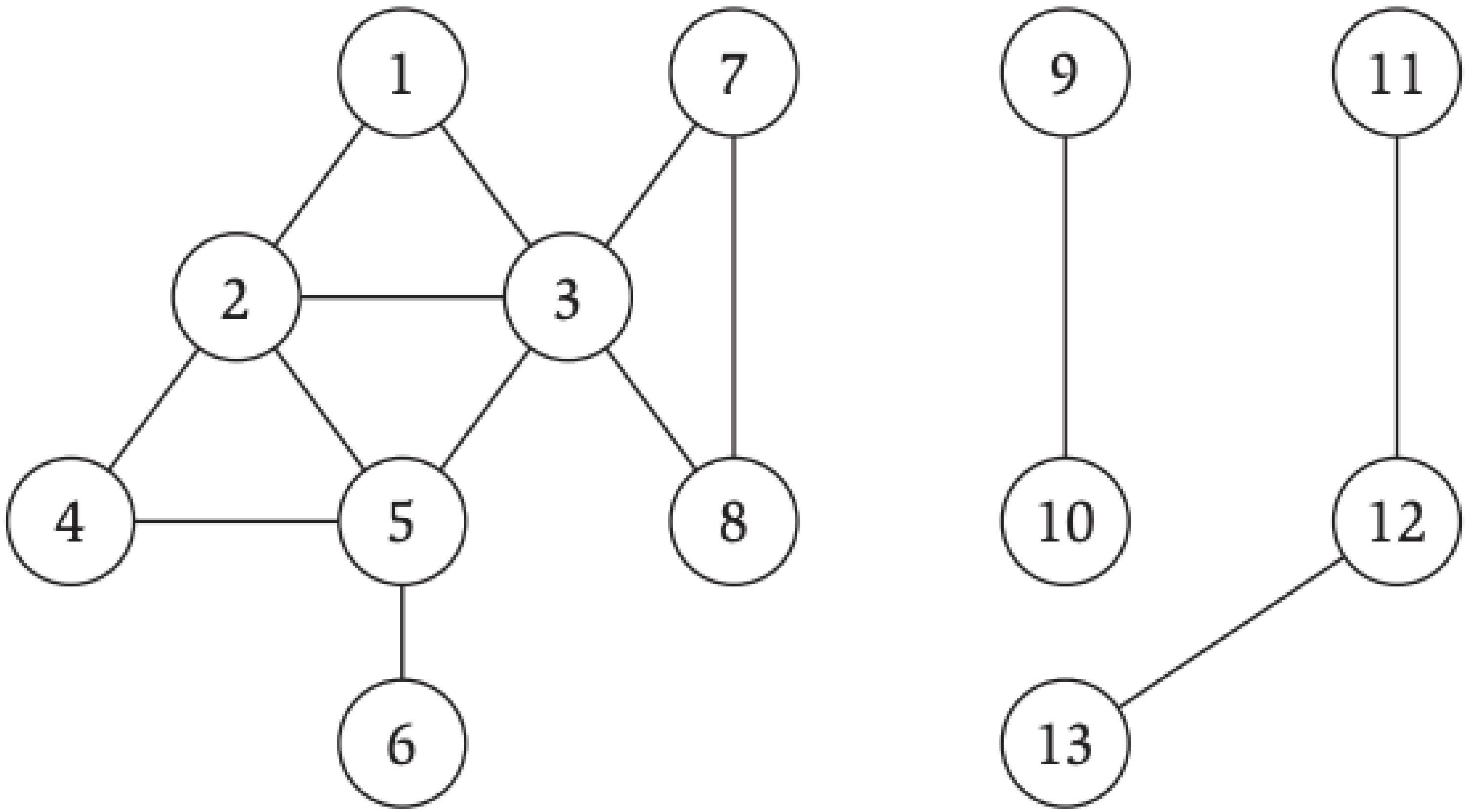
Another Connectivity Problem

Observation: If we solve the graph connectivity problem, then we can solve the s-t connectivity problem by checking if t is in T.



Breadth First Search

Q: How do you find all nodes reachable from node 1?

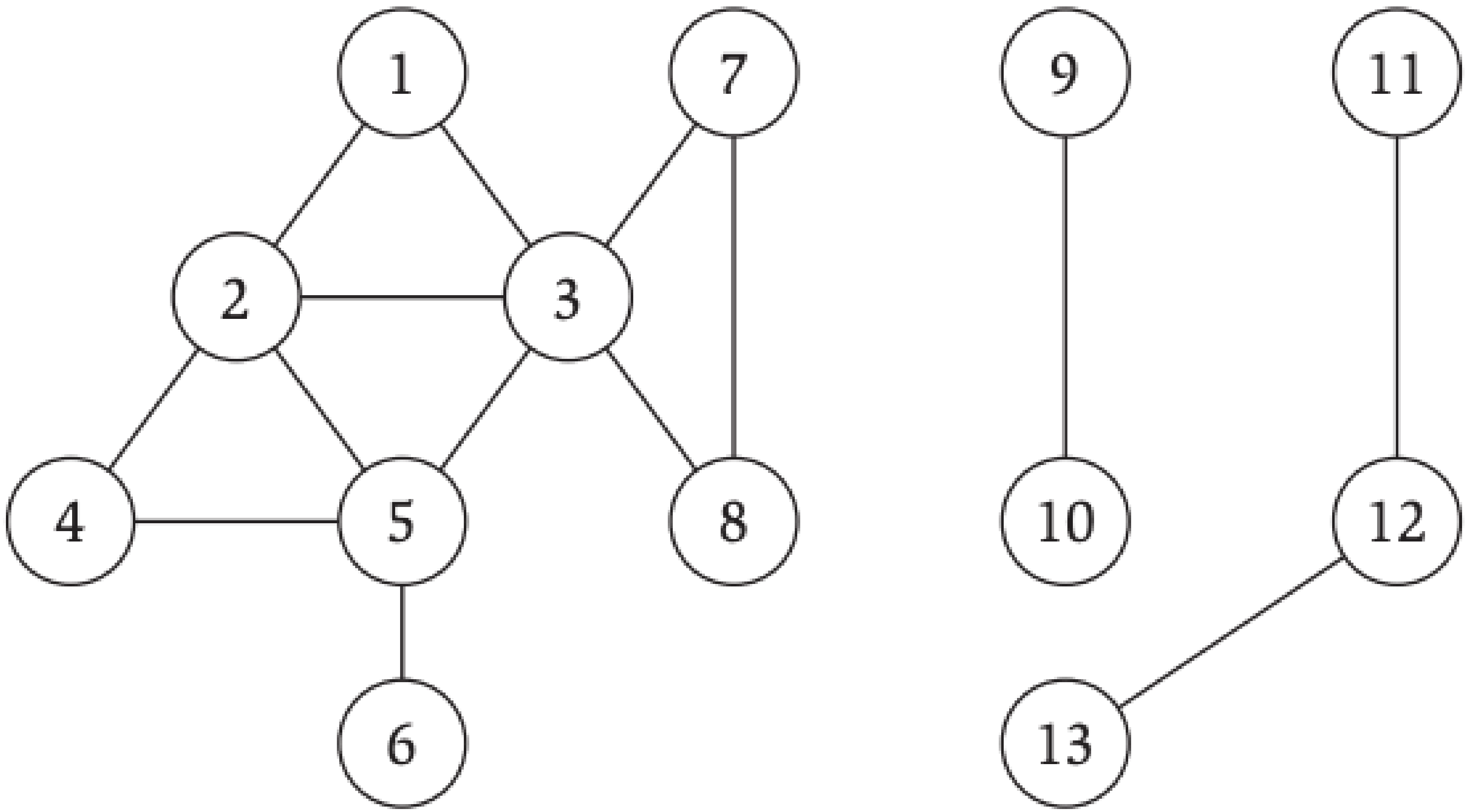


t

Breadth First Search

Idea:

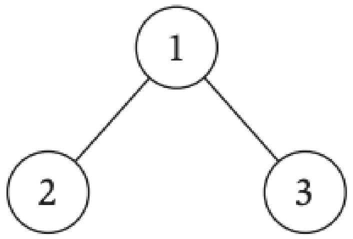
- What can 1 reach?
- What can those nodes reach?
- What can those nodes reach?
- ….

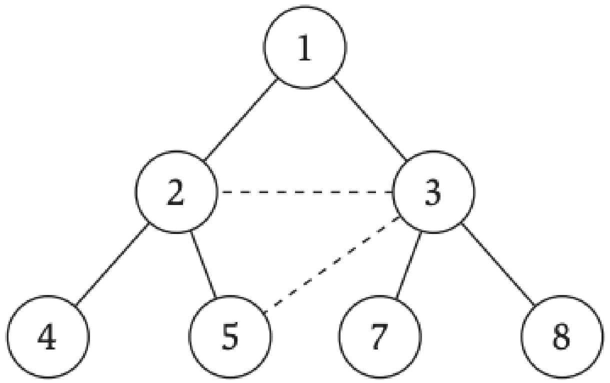


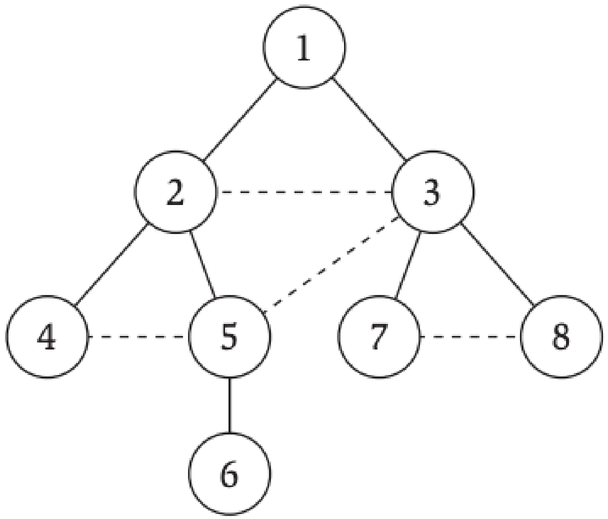
t

Breadth First Search (Layers)

- $L_0 = s$
- $L_1 =$ neighbors of L_0.
- $L_2 =$ neighbors of L_1 that are not in L_0.
- $L_{\mathrm{i}} =$ neighbors of $L_{\mathrm{i}-1}$ that are not in previous layer.







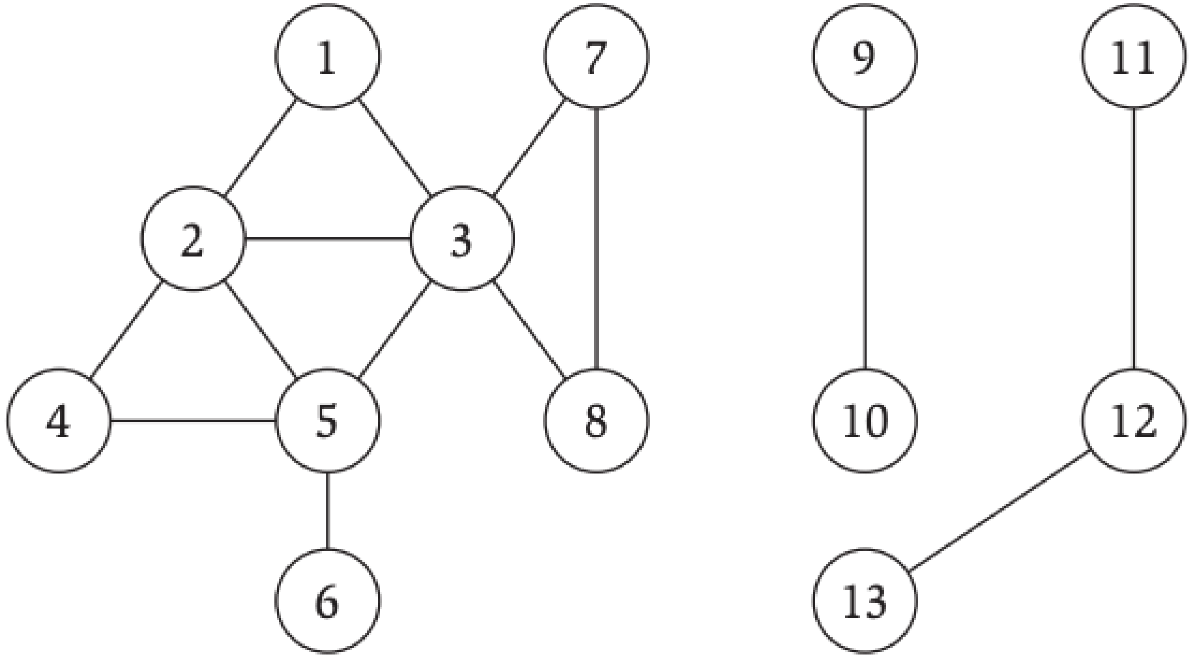
Breadth First Search ("Algorithm")

- Input: $G = (V, E)$ and $s \in V$
- Let $L_0 = \{s\}$
- Assume $L_0, \dots, L_i$ have been constructed:
  - Let $L_{\mathrm{i}+1}$ be nodes do not appear in $L_0, \dots, L_i$ and have an edge to L_i.
  - If $L_{\mathrm{i}+1}$ is empty, stop.
- Return all layers.

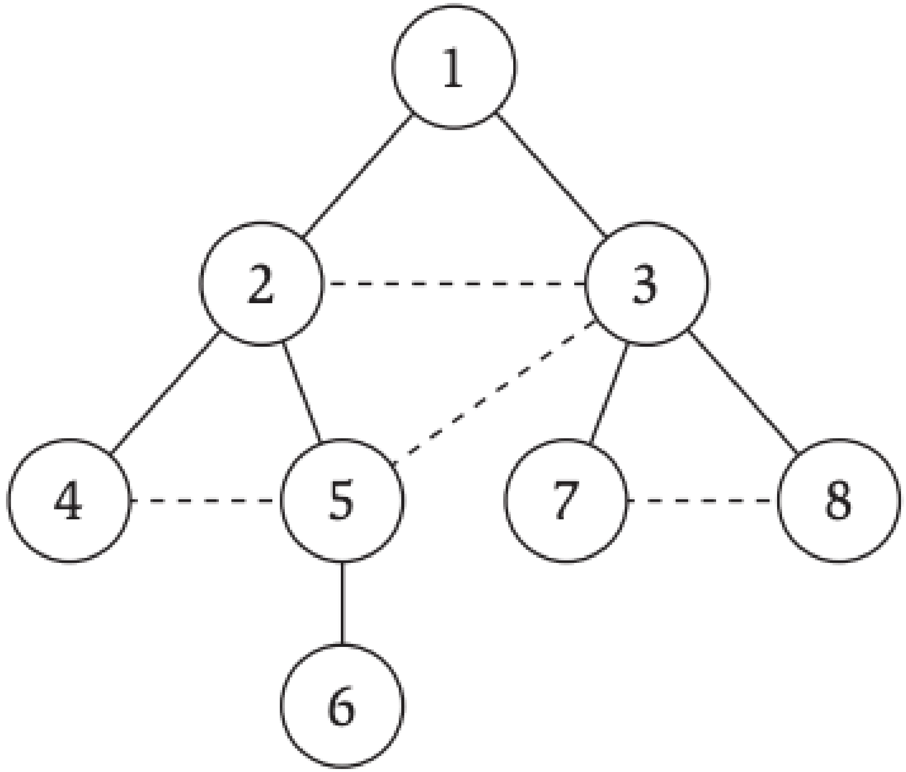
Breadth First Search (Properties)

- For each j ≥ 0, layer L_j produced by BFS consists of all nodes at distance j from s.
- There is a path from s to t if and only if t appears in some layer.
- For any $\{\mathrm{u}, \mathrm{v}\} \in \mathrm{E}$, if $u \in L_i$ and $v \in L_j$ then i and j differ by at most 1.
- You can think of the output as a tree! We call this the **BFS (discovery) Tree**.

Breadth First Search (Tree)



Original Graph



Search Tree