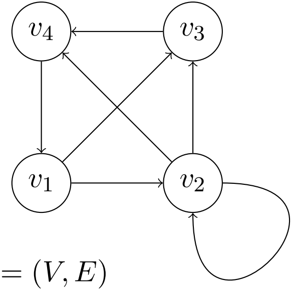
CS170: Discrete Methods in Computer Science
Spring 2025
Basics of Graph Theory

Instructor: Shaddin Dughmi[1]

[1] These slides adapt some content from similar slides by Aaron Cote.

Outline

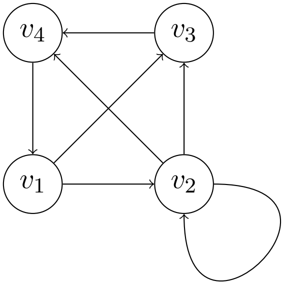
Definition of Directed Graph (Digraph)



- A Digraph is $G = (V, E)$
- V is a set of vertices or nodes
- $E \subseteq V \times V$ is a set of edges or arcs
  - An edge $e \in E$ from u to v denoted (u, v) or $u \to v$
  - Sometimes we allow self-loops $u \to u$
  - Rarely, allow parallel edges (E is a multiset). Gives multigraphs.
  - Usually disallow both, in which case we have simple graphs
- Conventionally $n = |V|$, $m = |E|$.
- In-degree $\deg^-(v)$ is number of edges entering v.
- Out-degree $\deg^+(v)$ is number of edges leaving v.

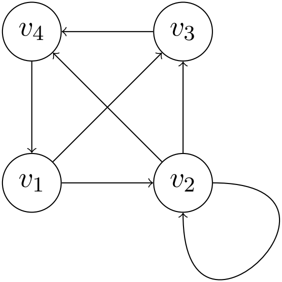
Hand-Shaking Lemma for Digraphs

$$\sum_{v \in V} \deg^{-}(v) = \sum_{v \in V} \deg^{+}(v) = m$$



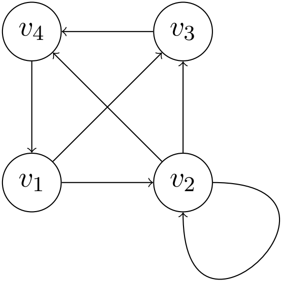
Walks, Paths, etc

- A walk is a sequence of nodes $u_1, u_2, \ldots, u_k$ such that $u_i \to u_{i+1}$ for each i
  - Nodes and edges not necessarily distinct
  - Length is $k - 1$: number of “hops”
- A path is a walk where all nodes are distinct
- A circuit is a walk with $u_1 = u_k$
- A cycle is a circuit with $u_1, \ldots, u_{k-1}$ distinct



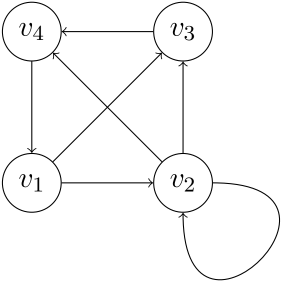
Walks, Paths, etc

- A walk is a sequence of nodes $u_1, u_2, \ldots, u_k$ such that $u_i \to u_{i+1}$ for each i
  - Nodes and edges not necessarily distinct
  - Length is $k - 1$: number of "hops"
- A path is a walk where all nodes are distinct
- A circuit is a walk with $u_1 = u_k$
- A cycle is a circuit with $u_1, \ldots, u_{k-1}$ distinct
- Note: Can turn any walk into a path, and any circuit into a cycle, by "skipping" intermediate cycles



Walks, Paths, etc

- A walk is a sequence of nodes $u_1, u_2, \ldots, u_k$ such that $u_i \to u_{i+1}$ for each i
  - Nodes and edges not necessarily distinct
  - Length is $k - 1$: number of "hops"
- A path is a walk where all nodes are distinct
- A circuit is a walk with $u_1 = u_k$
- A cycle is a circuit with $u_1, \ldots, u_{k-1}$ distinct
- Note: Can turn any walk into a path, and any circuit into a cycle, by "skipping" intermediate cycles
- Recall: A DAG is a Digraph with no cycles/circuits.

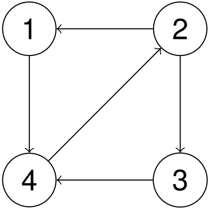


Shades of Connectivity

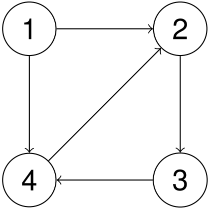
For a digraph $G = (V, E)$, we say it is:

- Strongly Connected if for any $u, v \in V$ there is a path from u to v, and a path from v to u.
- Unilaterally Connected if for any $u, v \in V$ there either path from u to v or a path from v to u.
- Weakly Connected if for any $u, v \in V$ you can get from u to v by ignoring edge directions.
- Disconnected (a.k.a. unconnected) otherwise

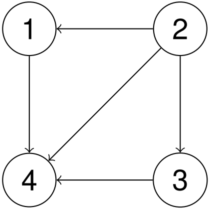
Shades of Connectivity: Examples



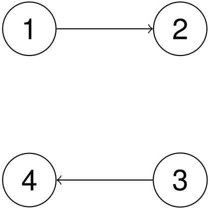
Strongly Connected



Unilaterally Connected



Weakly Connected



Disconnected

Subgraphs and Minors

- Edge deletion: $G - e$ (or $G \setminus e$) is the graph resulting from removing edge e from G
- Node deletion: $G - v$ (or $G \setminus v$) is the graph resulting from removing node v from G, as well as all it's incident edges

Subgraphs and Minors

- Edge deletion: $G - e$ (or $G \setminus e$) is the graph resulting from removing edge e from G
- Node deletion: $G - v$ (or $G \setminus v$) is the graph resulting from removing node v from G, as well as all it's incident edges
- $G' = (V', E')$ is a subgraph of $G = (V, E)$ if you can get it from G by a sequence of node and/or edge deletions
  - Equivalently: $V' \subseteq V$, $E' \subseteq E$, and G' is a graph.
- G' is the subgraph of G induced by V' if it has all the edges in G between nodes in V' (i.e. $E' = E \bigcap (V' \times V')$).
  - Equivalently: Result of deleting nodes $V - V'$ from G, in any order.

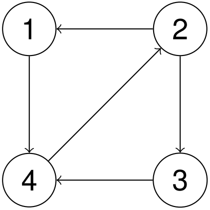
Subgraphs and Minors

- Edge deletion: $G - e$ (or $G \setminus e$) is the graph resulting from removing edge e from G
- Node deletion: $G - v$ (or $G \setminus v$) is the graph resulting from removing node v from G, as well as all it's incident edges
- $G' = (V', E')$ is a subgraph of $G = (V, E)$ if you can get it from G by a sequence of node and/or edge deletions
  - Equivalently: $V' \subseteq V$, $E' \subseteq E$, and G' is a graph.
- G' is the subgraph of G induced by V' if it has all the edges in G between nodes in V' (i.e. $E' = E \bigcap (V' \times V')$).
  - Equivalently: Result of deleting nodes $V - V'$ from G, in any order.
- Edge contraction: G/e for $e = (u, v)$ removes e, and combines u and v into one node with both their edges.
  - Not a subgraph of G.

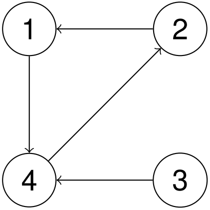
Subgraphs and Minors

- Edge deletion: $G - e$ (or $G \setminus e$) is the graph resulting from removing edge e from G
- Node deletion: $G - v$ (or $G \setminus v$) is the graph resulting from removing node v from G, as well as all it's incident edges
- $G' = (V', E')$ is a subgraph of $G = (V, E)$ if you can get it from G by a sequence of node and/or edge deletions
  - Equivalently: $V' \subseteq V$, $E' \subseteq E$, and G' is a graph.
- G' is the subgraph of G induced by V' if it has all the edges in G between nodes in V' (i.e. $E' = E \bigcap (V' \times V')$).
  - Equivalently: Result of deleting nodes $V - V'$ from G, in any order.
- Edge contraction: G/e for $e = (u, v)$ removes e, and combines u and v into one node with both their edges.
  - Not a subgraph of G.
- A Minor of G is any graph you can get from G by deletions and contractions.
  - Every subgraph is a minor, but not every minor is a subgraph

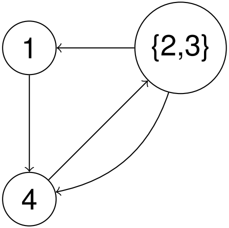
Subgraphs and Minors: Examples



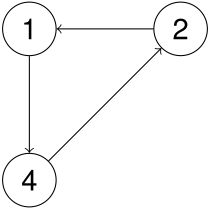
Graph G



Delete $(2,3)$



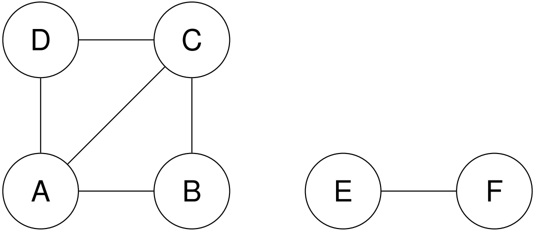
Contract $(2,3)$



Delete 3
or, induced by $\{1,2,4\}$

Outline

Undirected Graphs

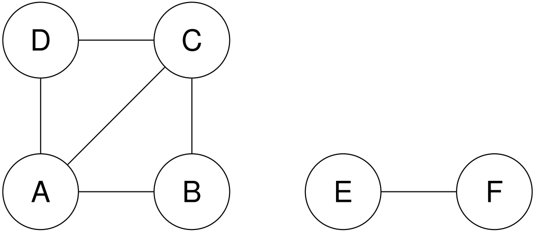


- $G = (V, E)$, same as directed graphs, but we ignore edge direction.
- $E \subseteq V \times V$ for convenience, though we dont distinguish between (u, v) and (v, u).
- $\deg(v)$ is number of edges with v as an endpoint.
- The neighbors of $u \in V$ are the nodes sharing an edge with u.

Hand-Shaking Lemma for Undirected Graphs

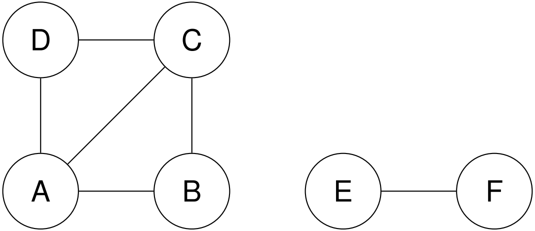
$$\sum_{v \in V} \deg(v) = 2m$$

Notable corollary: There is an even number of nodes with odd degree



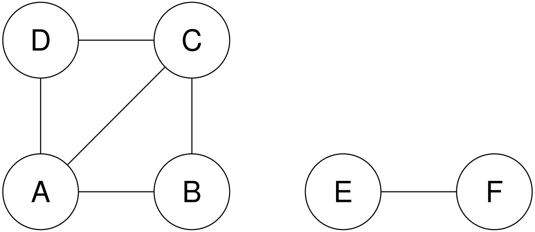
Walks, Paths, etc

- Same definition of walks, paths, cycles, circuits as in directed graphs, except each edge is interepreted as bidirectional
- For $(u, v) \in E$, can go from u to v or from v to u in one hop.



Connectivity

- No more “shades” of connectivity
- A graph is either connected or disconnected
- Connected: There is a path between any pair of nodes.
- For undirected graphs that are disconnected, can define a relation based on who can reach whom:
  - v is reachable from u if there is a path from u to v
- Reachability is an equivalence relation
- Equivalence classes are called connected components of G
  - These are the maximal connected subgraphs of G

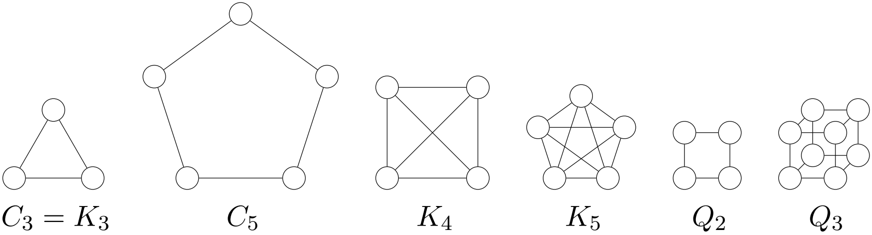


Outline

Simple Classes of Undirected Graphs

These are special classes of undirected graphs with one graph per number of nodes n

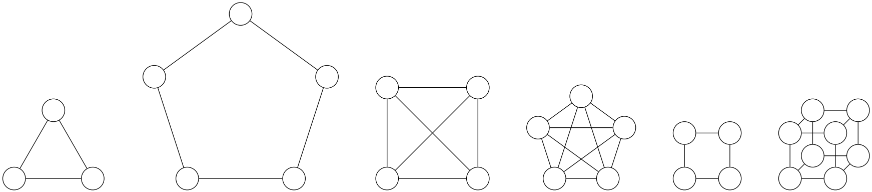
- Cycle graph C_n: An cycle on n nodes
- Complete Graph K_n: A graph on n nodes with an edge between every pair of distinct nodes.
- Hypercube graph Q_n: A graph with 2^n nodes representing the n dimensional hypercube (line, square, cube, etc).



Simple Classes of Undirected Graphs

These are special classes of undirected graphs with one graph per number of nodes n

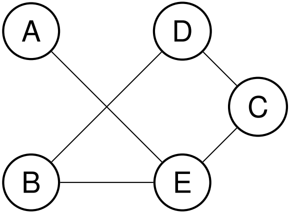
- Cycle graph C_n: An cycle on n nodes
- Complete Graph K_n: A graph on n nodes with an edge between every pair of distinct nodes.
- Hypercube graph Q_n: A graph with 2^n nodes representing the n dimensional hypercube (line, square, cube, etc).



$C_3 = K_3$ C_5 K_4 K_5 Q_2 Q_3

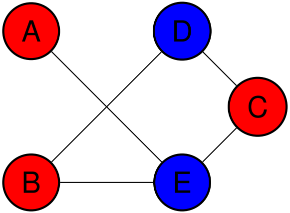
Note: There is a natural directed version of cycle and complete graphs.

Bipartite Graphs



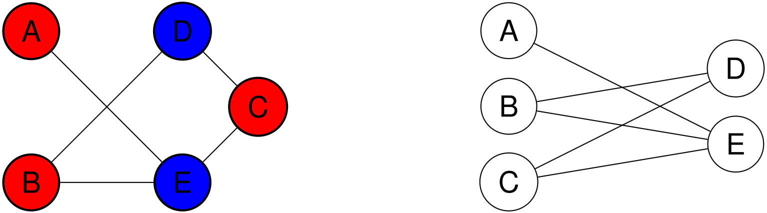
- An undirected graph $G = (V, E)$ is bipartite if we can color its nodes with two colors (e.g. red and blue) such that every edge is bichromatic (one endpoint of each color)

Bipartite Graphs



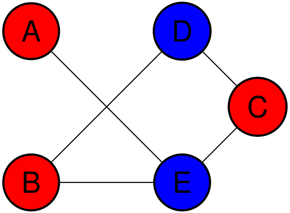
- An undirected graph $G = (V, E)$ is bipartite if we can color its nodes with two colors (e.g. red and blue) such that every edge is bichromatic (one endpoint of each color)

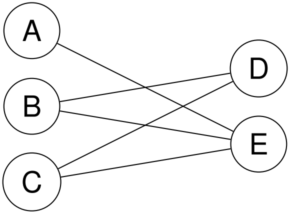
Bipartite Graphs



- An undirected graph $G = (V, E)$ is bipartite if we can color its nodes with two colors (e.g. red and blue) such that every edge is bichromatic (one endpoint of each color)
- Sometimes, we know upfront which nodes have each color, and we draw one color on the left and the other on the right

Bipartite Graphs

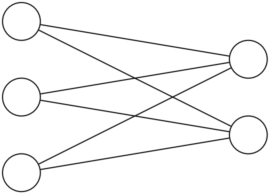




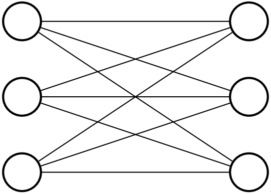
- An undirected graph $G = (V, E)$ is bipartite if we can color its nodes with two colors (e.g. red and blue) such that every edge is bichromatic (one endpoint of each color)
- Sometimes, we know upfront which nodes have each color, and we draw one color on the left and the other on the right
- Fact: A graph is bipartite iff it has no odd cycles
    - Ponder: Why is this necessary? Sufficient?

Bipartite Graphs

- Bipartite graphs are very important in CS to model relationships between two different sorts of objects
  - E.g. buyers and sellers, students and courses, applicants and jobs, inputs and outputs of a function, ...
  - Important algorithmic problem: bipartite matching.
- An $m \times n$ bipartite graph is one with m nodes on the left, n on the right, and edges only between the left and the right.
- The complete bipartite graph $K_{m,n}$ includes every edge between left and right.
  - mn edges total



$K_{3,2}$



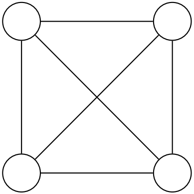
$K_{3,3}$

Planar Graphs

- A graph is planar if you can draw it in the plane without crossing edges.
- Note: A graph can be planar even if the drawing you have in front of you has crossing edges.

Planar Graphs

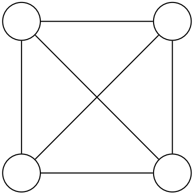
- A graph is planar if you can draw it in the plane without crossing edges.
- Note: A graph can be planar even if the drawing you have in front of you has crossing edges.



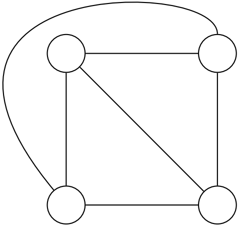
Is this planar?

Planar Graphs

- A graph is planar if you can draw it in the plane without crossing edges.
- Note: A graph can be planar even if the drawing you have in front of you has crossing edges.



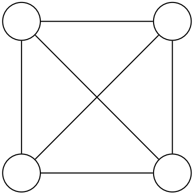
Is this planar?



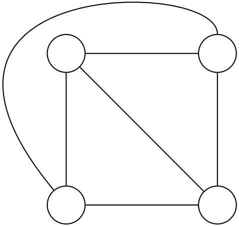
Yes!

Planar Graphs

- A graph is planar if you can draw it in the plane without crossing edges.
- Note: A graph can be planar even if the drawing you have in front of you has crossing edges.



Is this planar?

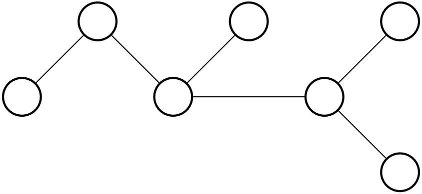


Yes!

Kuratowski's Theorem

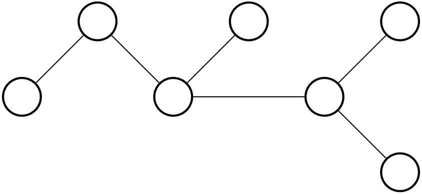
A graph is planar if and only if it excludes K_5 and $K_{3\times3}$ as a minor!

Trees



- An undirected graph is a tree if it is connected and acyclic
- Trees are very important in computer science
  - e.g. binary search trees, heaps, representing heirarchal data, decision trees, game trees, parse trees, spanning trees, space partitioning, compression, . . .

Trees



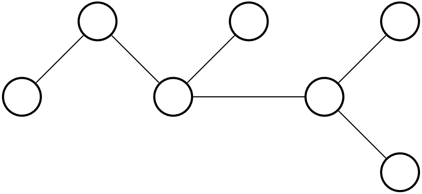
There are three equivalent definitions, per theorem below

Theorem

For an undirected graph G on n nodes, any two of the following imply the third

1. G is connected
2. G is acyclic
3. G has $n-1$ edges

Trees



There are three equivalent definitions, per theorem below

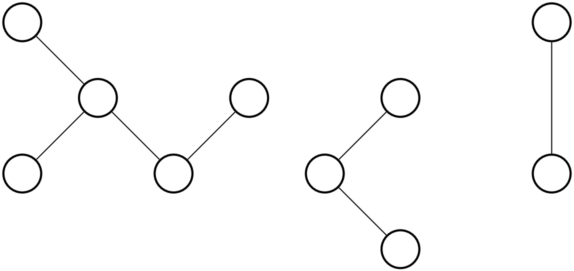
Theorem

For an undirected graph G on n nodes, any two of the following imply the third

1. G is connected
2. G is acyclic
3. G has $n-1$ edges

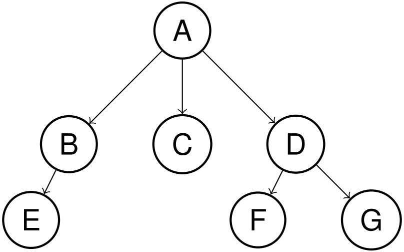
Fact: In a tree, for each pair of nodes u and v there is exactly one path from u to v. (Why?)

Forests



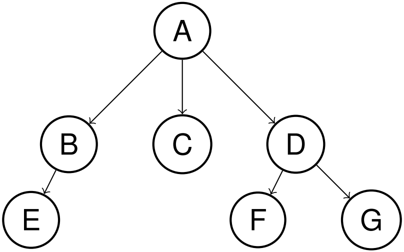
- An undirected graph which is acyclic (but not necessarily connected) is called a forest.
- Why?
  - It's connected components are connected and acyclic, i.e., trees
  - So it is the disjoint union of trees, i.e. a “forest”

Rooted Trees



- In many applications, it makes sense to pick a “root” for the tree, and direct all edges away from the root
- We call these directed graphs rooted trees, and draw them either top down or bottom up.
- For $u \to v$, we say v is a child of u, and u is the parent of v
- The root has zero or more children, but no parent.
- Every other node has one parent and zero or more children.

Rooted Trees



- A node with no children is called a leaf
- Nodes other than the root and the leaves called internal nodes.
- The depth of the tree is the maximum distance from root to leaf
- A tree is binary if each node has at most 2 children. d-ary if at most d children.
- For a node u, the subtree rooted as u is the subgraph induced by u and its descendents (nodes reachable from u by directed edges).

Outline

Graph Isomorphism

- We consider two graphs “the same” even if drawn differently.
  - Many different ways to embed the same graph into the plane
- Often, we also don’t care about what names you give the nodes and edges
- Isomorphism captures what it means for two graphs to be “the same”, disregarding names of nodes and edges, and without regard to how you draw them.

Graph Isomorphism

Formal Definition

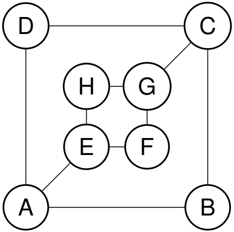
We say graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there is a bijection $f : V \to V'$ such that

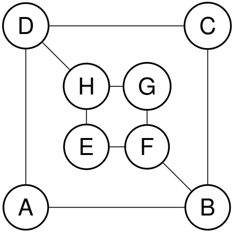
$$(u, v) \in E \iff (f(u), f(v)) \in E'$$

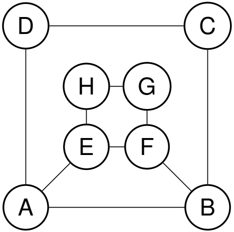
for all $u, v \in V$

- Gives an equivalence relation on graphs
- Same for directed and undirected graphs

Which of the following are isomorphic?







Outline

Graph Proofs

- Proofs involving graphs are just like proofs involving any other sort of mathematical object
- E.g. Proofs of handshake lemmas.
- Let's see a few more.

Two nodes must have the same degree

Claim

A simple undirected graph G with $n \geq 2$ has two nodes of the same degree.

Two nodes must have the same degree

Claim

A simple undirected graph G with $n \geq 2$ has two nodes of the same degree.

Proof

- If G has no edges, two nodes have degree 0 and we are done. Otherwise,
- There is a connected component G' of G with $n' \geq 2$ nodes.
- Each node in G' has degree between 1 and $n' - 1$ (inclusive).
- Letting the n' nodes of G' be pigeons and their degrees be the pigeonholes, two nodes have the same degree.

Claim

If an undirected simple graph has a circuit of odd length, then it has a cycle of odd length.

Claim

If an undirected simple graph has a circuit of odd length, then it has a cycle of odd length.

Proof

- By strong induction on length k of the circuit
- Base case $k \leq 3$: Only such circuits are cycles of length 3, so true.
- Assume true for odd lengths $\leq k$, and consider circuit C of odd length $k + 2$
- If C is a cycle we are done. Otherwise, contains smaller circuit C'.
- One of C' or $C \setminus C'$ is has odd length $\leq k$.
- Invoke inductive hypothesis on whichever one that is.

Claim

A simple undirected graph is bipartite iff it has no odd cycles.

Claim

A simple undirected graph is bipartite iff it has no odd cycles.

Need to prove necessity and sufficiency

Proof: Necessity

- No bichromatic coloring for an odd cycle (since colors alternate).
- Therefore, no such coloring for graph including odd cycle.

Claim

A simple undirected graph is bipartite iff it has no odd cycles.

Need to prove necessity and sufficiency

Proof: Sufficiency

- Suppose there are no odd cycles
- Let $u \in V$ be arbitrary.
- Color $v \in V$ red if distance $d(u, v)$ from u even, blue otherwise.
- Consider $(v_1, v_2) \in E$.
- Note that $|d(u, v_1) - d(u, v_2)| \leq 1$ since they share an edge.

Claim

A simple undirected graph is bipartite iff it has no odd cycles.

Need to prove necessity and sufficiency

Proof: Sufficiency

- Suppose there are no odd cycles
- Let $u \in V$ be arbitrary.
- Color $v \in V$ red if distance $d(u, v)$ from u even, blue otherwise.
- Consider $(v_1, v_2) \in E$.
- Note that $|d(u, v_1) - d(u, v_2)| \leq 1$ since they share an edge.
- If $|d(u, v_1) - d(u, v_2)| = 1$ then one is even and one is odd, so different colors.

Claim

A simple undirected graph is bipartite iff it has no odd cycles.

Need to prove necessity and sufficiency

Proof: Sufficiency

- Suppose there are no odd cycles
- Let $u \in V$ be arbitrary.
- Color $v \in V$ red if distance $d(u, v)$ from u even, blue otherwise.
- Consider $(v_1, v_2) \in E$.
- Note that $|d(u, v_1) - d(u, v_2)| \leq 1$ since they share an edge.
- If $|d(u, v_1) - d(u, v_2)| = 1$ then one is even and one is odd, so different colors.
- If $|d(u, v_1) - d(u, v_2)| = 0$, i.e. $d(u, v_1) = d(u, v_2) := d$, then
  - There is a circuit involving u, v_1, v_2 of odd length $2d + 1$
  - Therefore, there is an odd cycle (See previous claim)
  - We assumed there are no odd cycles, so this case does not occur.

Claim

For an undirected graph G with n nodes, m edges, and c connected components, the following holds: $m \geq n - c$

Claim

For an undirected graph G with n nodes, m edges, and c connected components, the following holds: $m \geq n - c$

Proof

- We prove this by induction on n. Base case $n = 1$ is trivial.
- Inductive Hyp.: Suppose it holds for graphs with $n - 1$ nodes.

Claim

For an undirected graph G with n nodes, m edges, and c connected components, the following holds: $m \geq n - c$

Proof

- We prove this by induction on n. Base case $n = 1$ is trivial.
- Inductive Hyp.: Suppose it holds for graphs with $n - 1$ nodes.
- Consider G with n nodes. For arbitrary vertex v, let $G' = G - v$.

Claim

For an undirected graph G with n nodes, m edges, and c connected components, the following holds: $m \geq n - c$

Proof

- We prove this by induction on n. Base case $n = 1$ is trivial.
- Inductive Hyp.: Suppose it holds for graphs with $n - 1$ nodes.
- Consider G with n nodes. For arbitrary vertex v, let $G' = G - v$.
- G' has $n' = n - 1$ nodes, m' edges, and c' connected components.
- By inductive hypothesis, we have $m' \geq n' - c'$.

Claim

For an undirected graph G with n nodes, m edges, and c connected components, the following holds: $m \geq n - c$

Proof

- We prove this by induction on n. Base case $n = 1$ is trivial.
- Inductive Hyp.: Suppose it holds for graphs with $n - 1$ nodes.
- Consider G with n nodes. For arbitrary vertex v, let $G' = G - v$.
- G' has $n' = n - 1$ nodes, m' edges, and c' connected components.
- By inductive hypothesis, we have $m' \geq n' - c'$.
- Let k be the number of components of G' to which v has an edge.
- $m \geq m' + k$, since v has at least one edge to each of them.
- v merges those k components, so $c = c' - (k - 1)$.

Claim

For an undirected graph G with n nodes, m edges, and c connected components, the following holds: $m \geq n - c$

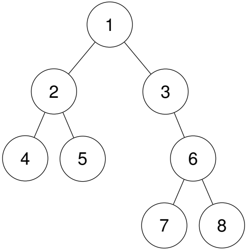
Proof

- We prove this by induction on n. Base case $n = 1$ is trivial.
- Inductive Hyp.: Suppose it holds for graphs with $n - 1$ nodes.
- Consider G with n nodes. For arbitrary vertex v, let $G' = G - v$.
- G' has $n' = n - 1$ nodes, m' edges, and c' connected components.
- By inductive hypothesis, we have $m' \geq n' - c'$.
- Let k be the number of components of G' to which v has an edge.
- $m \geq m' + k$, since v has at least one edge to each of them.
- v merges those k components, so $c = c' - (k - 1)$.
- We get:

$$m \geq m' + k \geq n' - c' + k = (n - 1) - (c + k - 1) + k = n - c$$

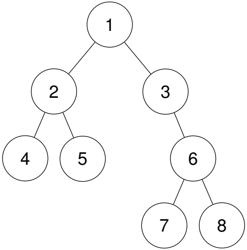
Outline

Depth-first Tree Traversal



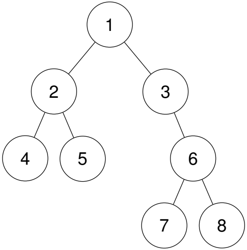
- Depth-first: "Go deep" first, backtrack when branch exhausted
- Three variations: Pre-order, post-order, in-order

Depth-first Tree Traversal



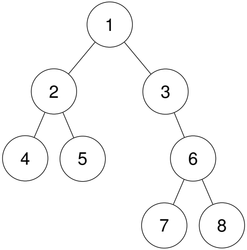
- Depth-first: "Go deep" first, backtrack when branch exhausted
- Three variations: Pre-order, post-order, in-order
- Pre-order: parent then children
  - 1,2,4,5,3,6,7,8
  - copy, print in prefix notation, topological sort

Depth-first Tree Traversal



- Depth-first: “Go deep” first, backtrack when branch exhausted
- Three variations: Pre-order, post-order, in-order
- Pre-order: parent then children
  - 1,2,4,5,3,6,7,8
  - copy, print in prefix notation, topological sort
- Post-order: children then parent
  - 4,5,2,7,8,6,3,1
  - delete, evaluate math expression, hierarchal tasks, topological sort.

Depth-first Tree Traversal



- Depth-first: "Go deep" first, backtrack when branch exhausted
- Three variations: Pre-order, post-order, in-order
- Pre-order: parent then children
  - 1,2,4,5,3,6,7,8
  - copy, print in prefix notation, topological sort
- Post-order: children then parent
  - 4,5,2,7,8,6,3,1
  - delete, evaluate math expression, hierarchal tasks, topological sort.
- In-order (for binary trees): left child, parent, right child.
  - 4,2,5,1,3,7,6,8
  - print in infix notation, sort BST.

Pseudocode for Tree Traversal

Call the following recursive function with $u = r$, where r is the root.

Traverse-inorder(u):

- If (left(u) $\neq$ null) then Traverse-inorder(left(u))
- visit(u) (i.e., “do something” for u)
- If (right(u) $\neq$ null) then Traverse-inorder(right(u))

Pseudocode for Tree Traversal

Call the following recursive function with $u = r$, where r is the root.

Traverse-inorder(u):

- If (left(u) $\neq$ null) then Traverse-inorder(left(u))
- visit(u) (i.e., "do something" for u)
- If (right(u) $\neq$ null) then Traverse-inorder(right(u))

How do you change this to pre-order or post-order?

Pseudocode for Tree Traversal

Call the following recursive function with $u = r$, where r is the root.

Traverse-inorder(u):

- If (left(u) $\neq$ null) then Traverse-inorder(left(u))
- visit(u) (i.e., "do something" for u)
- If (right(u) $\neq$ null) then Traverse-inorder(right(u))

Runtime of Traverse(r)

If visiting takes $O(1)$, then $O(n)$.

Graph Traversal (a.k.a. Graph Search)

- Explore a directed or undirected graph $G = (V, E)$ starting from a node $s \in V$
- Goal is to "visit" each node u reachable from s
  - Do something for each such u, e.g. check if it's something we're searching for, add it to a list, etc.
- Two main algorithms: Depth-first search (DFS) and Breadth-first search (BFS)
- Both run in time $O(m + n)$
- In both cases, edges traversed form a tree (DFS Tree, BFS Tree), with various algorithmic applications

Depth First Search (DFS)

- Follow a path until you dead-end (i.e., go as deep as you can)
- Then backtrack
- Does not find shortest paths
- Admits a simple recursive implementation
- Memory efficient when the graph is “shallow” (no really long paths)
- Useful for some algorithmic applications (maze solving, bridge finding, etc)

Recursive implementation of DFS

Initialize visited$[v]$ = false for all nodes v, then invoke DFS(s)

DFS(u):

- visit(u)
- Set visited$[u]$ to true
- For each edge $u \to v$ with visited$[v]$ = false
  - DFS(v)

Example on board

Recursive implementation of DFS

Initialize visited$[v]$ = false for all nodes v, then invoke DFS(s)

DFS(u):

- visit(u)
- Set visited$[u]$ to true
- For each edge $u \to v$ with visited$[v]$ = false
  - DFS(v)

Example on board

Runtime

- Suppose visit(.) takes $O(1)$
- We visit each node, which takes $O(n)$
- For each u, we loop $\deg^+(u)$ edges
  - Sum of degrees is $O(m)$
- Total $O(m+n)$.

Iterative Implementation of DFS

The following implementation uses a stack datastructure (last-in first-out).

DFS-iter(s):

- Initialize empty stack T
- Set visited$[v]$ to false for all nodes.
- Push s onto T
- While T is non-empty
  - Pop a node u off T
  - if visited$[u]$ is false then
    - visit(u)
    - Set visited$[u]$ to true
    - For each edge $u \to v$, push v onto T

Iterative Implementation of DFS

The following implementation uses a stack datastructure (last-in first-out).

DFS-iter(s):

- Initialize empty stack T
- Set visited$[v]$ to false for all nodes.
- Push s onto T
- While T is non-empty
  - Pop a node u off T
  - if visited$[u]$ is false then
    - visit(u)
    - Set visited$[u]$ to true
    - For each edge $u \to v$, push v onto T

Runtime is $O(m + n)$, by same argument and the fact that push/pop take constant time.

Breadth First Search (BFS)

- Visit nodes in order of distance from the start node
- Finds shortest paths
- No simple recursive implementation
- Memory efficient when the graph is “skinny”
- Useful for some algorithmic applications: shortest paths and computing distances, finding nearby objects, checking bipartiteness, etc

Implementation of BFS

Same as iterative DFS, but with a Queue (first-in first-out).

BFS(s):

- Initialize empty queue Q
- Set visited$[v]$ to false for all nodes.
- Enqueue s onto Q
- While Q is non-empty
  - Dequeue a node u from Q
  - if visited$[u]$ is false then
    - visit(u)
    - Set visited$[u]$ to true
    - For each edge $u \to v$, enqueue v onto Q

Implementation of BFS

Same as iterative DFS, but with a Queue (first-in first-out).

BFS(s):

- Initialize empty queue Q
- Set visited$[v]$ to false for all nodes.
- Enqueue s onto Q
- While Q is non-empty
  - Dequeue a node u from Q
  - if visited$[u]$ is false then
    - visit(u)
    - Set visited$[u]$ to true
    - For each edge $u \to v$, enqueue v onto Q

Runtime is $O(m + n)$, by same argument and the fact that enqueue/dequeue take constant time.