Discrete Mathematics & Mathematical Reasoning
Chapter 11: Trees

Kousha Etessami

U. of Edinburgh, UK

A **tree** is a connected simple undirected graph with no simple circuits.
A **forest** is a (not necessarily connected) simple undirected graph with no simple circuits.

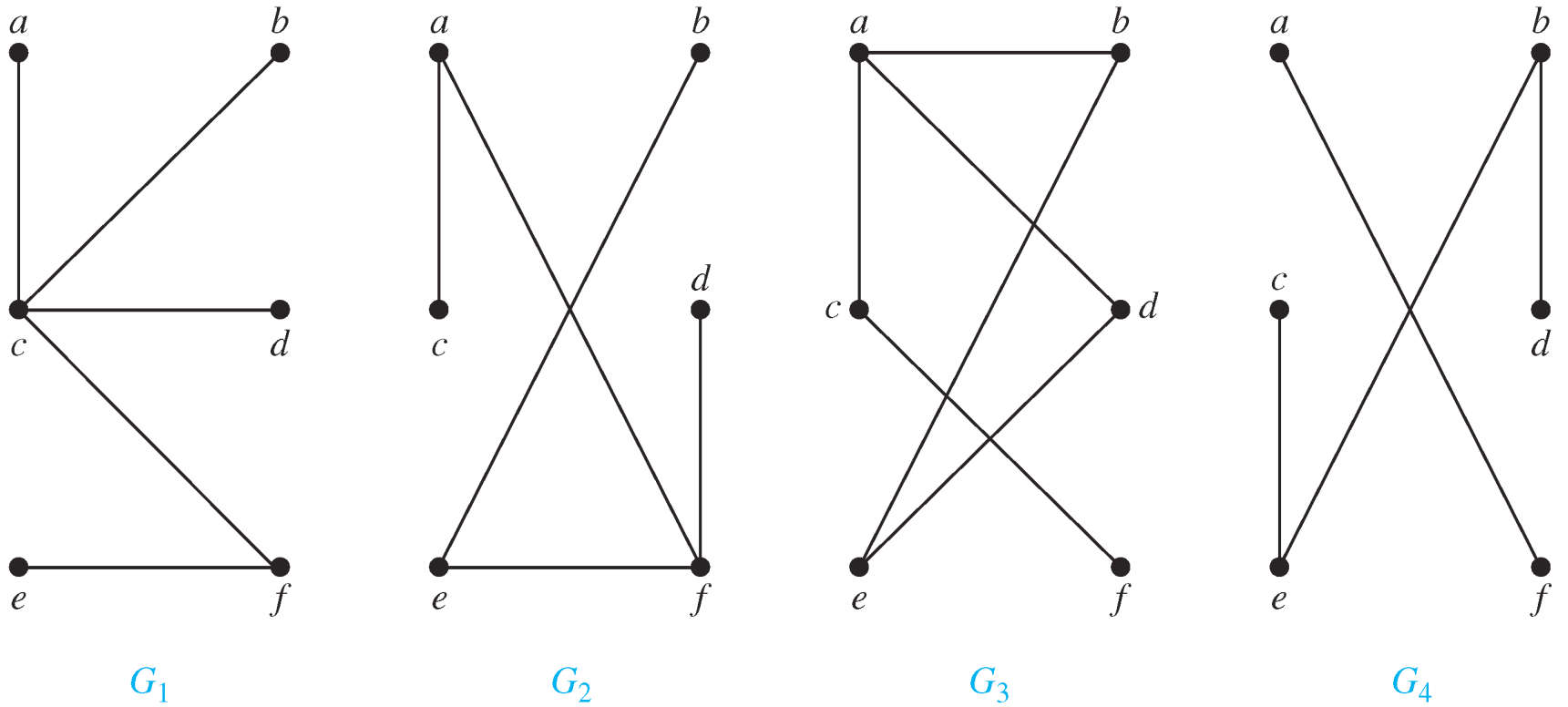


FIGURE 2 **Examples of Trees and Graphs That Are Not Trees.**

Some important facts about trees

Theorem 1: A graph G is a tree if and only if there is a unique simple (and tidy) path between any two vertices of G.

Proof: On the board. (Next slide provides written proof.) □

Theorem 2: Every tree, $T = (V, E)$ with $|V| \geq 2$, has at least two vertices that have degree $= 1$.

Proof: Take any **longest** simple path $x_0 \ldots x_m$ in T. Both x_0 and x_m must have degree 1: otherwise there's a longer path in T. □

Theorem 3: Every tree with n vertices has exactly $n - 1$ edges.

Proof: On the board. By induction on n. □

Proof of Theorem 1 about Trees

Suppose there are two distinct simple paths between vertices $u, v \in V$: $x_0 x_1 x_2 \dots x_n$ and $y_0 y_1 y_2 \dots y_m$.
Firstly, there must be some $i \geq 1$, such that $\forall 0 \leq k < i$, $x_k = y_k$, but such that $x_i \neq y_i$. (Why is this so?)
Furthermore, there must be a smallest $j \geq i$, such that either x_j appears in $y_i, \dots, y_m$, or such that y_j appears in $x_i \dots x_n$.
Suppose, without loss of generality, that this holds for some smallest $j \geq i$ and x_j. Then $x_j = y_r$, for some smallest $r \geq i$.
We claim that then the path $x_{i-1} x_i \dots x_j y_{r-1} y_{r-2} \dots y_i y_{i-1}$ must form a simple circuit, which contradicts the fact that G is a tree.
Note that by assumption $x_{i-1} = y_{i-1}$, so this is a circuit.
Furthermore, it is simple, because its edges are a disjoint union of edges from the x and y paths, because by construction none of the vertices $x_i, \dots, x_j$ occur in $y_i \dots y_{r-1}$, and $x_i \neq y_i$. □

Rooted Trees

A **rooted tree**, is a pair (T, r) where $T = (V, E)$ is a tree, and $r \in V$ is a chosen **root** vertex of the tree.

Often, the edges of a rooted tree (T,r) are viewed as being directed, such that for every vertex v the unique path from r to v is directed *away from* (or *towards*) r.

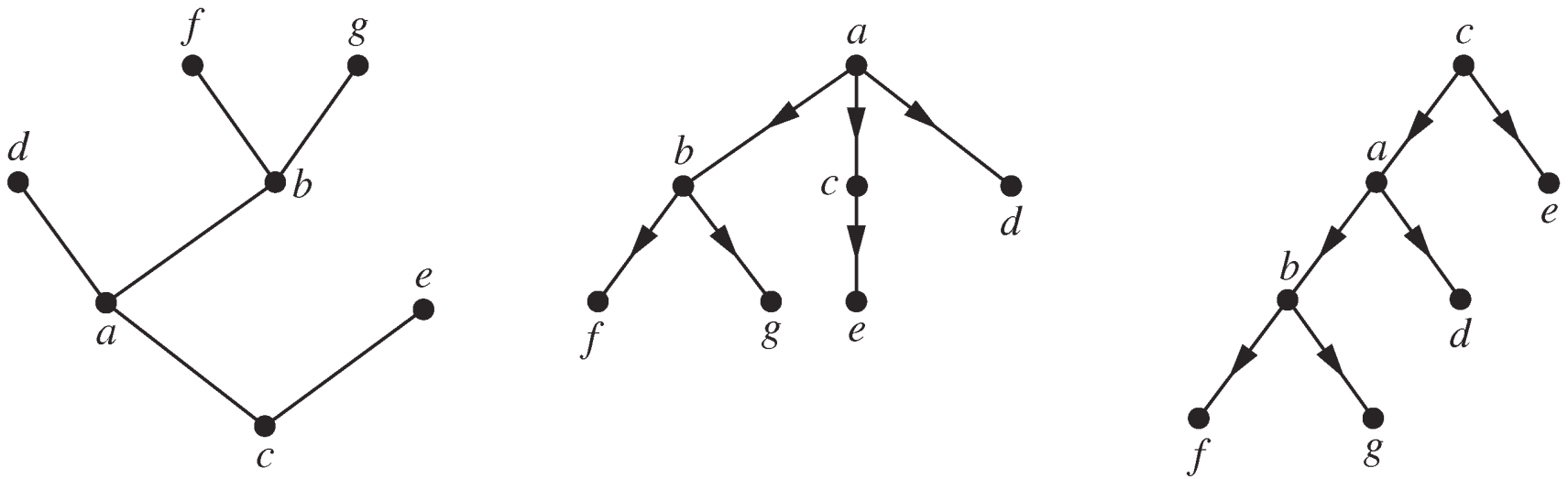
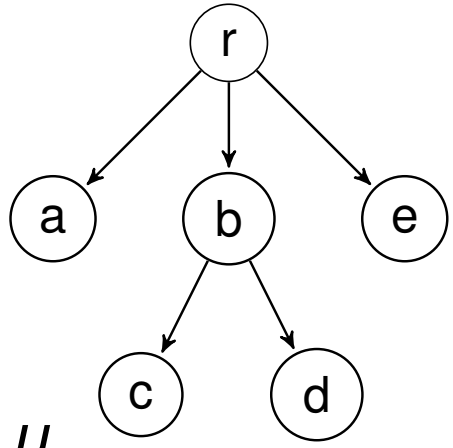


FIGURE 4 **A Tree and Rooted Trees Formed by Designating Two Different Roots.**

(In CS, rooted trees are typically drawn with root at the top.)

Terminology for rooted trees

For a rooted tree (T, r), with root r,



- For each node $v \neq r$ the **parent**, is the unique vertex u such that $(u, v) \in E$. v is then called a **child** of u. Two vertices with the same parent are called **siblings**.
- A **leaf** is a vertex with no children. Non-leaves are called **internal vertices**.
- The **height** of a rooted tree is the length of the longest directed path from the root to any leaf.
- The **ancestors** (**descendants**, respectively) of a vertex v are all vertices $u \neq v$ such that there is a directed path from u to v (from v to u, respectively).
- The **subtree** rooted at v, is the subgraph containing v and all its descendants, and all directed edges between them.

m-ary Trees

For $m \geq 1$, A rooted tree is called a ***m*-ary tree** if every internal node has at most m children.
It is called a **full *m*-ary tree** if every internal node has exactly m children.
An m-ary tree with $m = 2$ is called a **binary tree**.

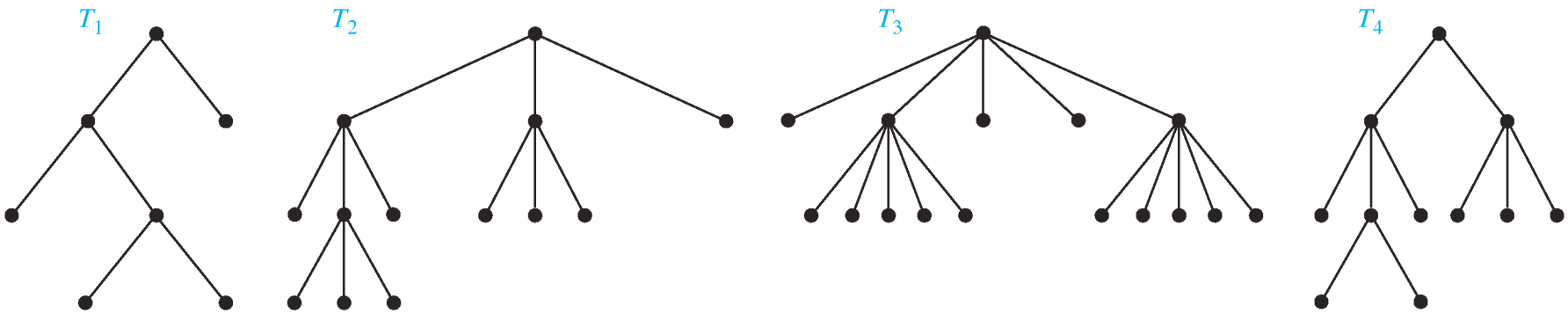


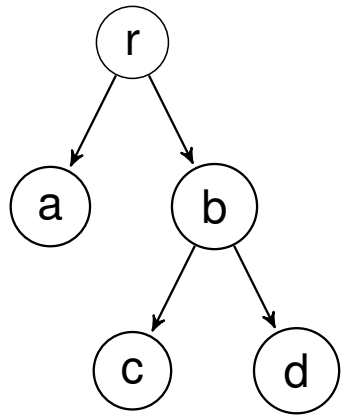
FIGURE 7 **Four Rooted Trees.**

Which one of these rooted trees is a (full) m-ary tree?

A rooted ordered tree is a rooted tree (T, r) where in addition the children of each internal vertex v are linearly ordered according to some ordering $\leq_v$.

When drawing the tree, we usually write ordered children (from least to greatest) from left to right.

If the rooted ordered tree is a binary tree, then the first child is called **left child** and the second child is called **right child**.



Note: rooted ordered trees are VERY COMMON in computer science applications: parse trees, XML documents, file directories, "decision trees", "game trees", ...

Counting vertices in m-ary trees

Theorem C1: For all $m \geq 1$, every full m-ary tree with i internal vertices has exactly $n = m \cdot i + 1$ vertices.

Proof: Every vertex other than the root is a child of an internal vertex. There are thus $m \cdot i$ such children, plus 1 root. □

Theorem C2: For all $m \geq 1$, a full m-ary tree with:

1. n vertices has $i = (n-1)/m$ internal vertices and $l = [(m-1)n + 1]/m$ leaves.
2. i internal vertices has $n = m \cdot i + 1$ vertices and $l = (m-1)i + 1$ leaves.
3. if $m \geq 2$, then if the m-ary tree has l leaves then it has $n = (ml - 1)/(m-1)$ vertices and $i = (l-1)/(m-1)$ internal vertices.

More counting for m-ary trees

Theorem C3: There are at most m^h leaves in an m-ary tree of height h.

Proof: By induction on $h \geq 0$. □

Theorem C4: If an m-ary tree has l leaves, and h is its height, then $h \geq \lceil \log_m l \rceil$.

Proof: Since $l \leq m^h$, we have $\log_m l \leq h$. But h in a non-negative integer, so $\lceil \log_m l \rceil \leq h$. □

Application: bounds for comparison-based sorting

Question: You have to sort a list of distinct unknown numbers: $a_1, \ldots, a_n$, using only the operation of comparing two numbers: $a_i \overset{?}{<} a_j$. How many comparisons do you need, in the worst case, in order to sort all the numbers correctly?

Application: bounds for comparison-based sorting

Question: You have to sort a list of distinct unknown numbers: $a_1, \ldots, a_n$, using only the operation of comparing two numbers: $a_i \overset{?}{<} a_j$. How many comparisons do you need, in the worst case, in order to sort all the numbers correctly?

Answer: Consider a binary **decision tree**, modeling the comparisons you make. There are $n!$ permutations of $a_1, \ldots, a_n$, so there are $n!$ possible fully sorted orderings. These constitute the leaves of your decision tree.
Since the decision tree is binary (2-ary), by Theorem C4 the height of the tree is $h \geq \log_2 n!$. But note that the height is the worst-case number of comparisons.
By Stirling's formula, $h \geq \log_2 \left(\frac{n}{e}\right)^n = \Omega(n \log_2 n)$. □

Spanning Trees of undirected graphs

For a simple undirected graph G, a **spanning tree** of G is a subgraph T of G such that T is a tree and T contains every vertex of G.

Theorem: Every connected graph G has a spanning tree.

Proof: While there is a circuit in G, remove one edge of the circuit. Repeat. Removing one edge of the circuit does not change connectivity, and eventually no circuits can remain (because there are only finitely many edges to be removed). So, the end result is a tree which is a subtree of G. □

Spanning Trees of undirected graphs

For a simple undirected graph G, a **spanning tree** of G is a subgraph T of G such that T is a tree and T contains every vertex of G.

Theorem: Every connected graph G has a spanning tree.

Proof: While there is a circuit in G, remove one edge of the circuit. Repeat. Removing one edge of the circuit does not change connectivity, and eventually no circuits can remain (because there are only finitely many edges to be removed). So, the end result is a tree which is a subtree of G. □

Question: Given a graph G, can we efficiently compute a spanning tree for G?

Spanning Trees of undirected graphs

For a simple undirected graph G, a **spanning tree** of G is a subgraph T of G such that T is a tree and T contains every vertex of G.

Theorem: Every connected graph G has a spanning tree.

Proof: While there is a circuit in G, remove one edge of the circuit. Repeat. Removing one edge of the circuit does not change connectivity, and eventually no circuits can remain (because there are only finitely many edges to be removed). So, the end result is a tree which is a subtree of G. □

Question: Given a graph G, can we efficiently compute a spanning tree for G?

Answer: Yes. Even for edge-weighted graphs, we can compute a **minimum-cost spanning tree** efficiently. (The cost of a spanning tree is the sum of its edge costs.)

Prim's algorithm for a minimum spanning tree

Input: Connected, edge-weighted, undirected graph
$$G = (V, E, w).$$

Output: A minimum-cost spanning tree T for G.

Algorithm:

Initialize: $T := \{e\}$, where e is a minimum-weight edge in E.
for $i := 1$ to $n - 2$ **do**
    Let $e' :=$ a minimum-weight edge incident to
        some vertex in T, and not forming a circuit
        if added to T;
    $T := T \cup \{e'\}$;
end for
Output the tree T.