

EE360C: Algorithms

Graphs

Part 2/3

Summer 2019

Department of Electrical and Computer Engineering
University of Texas at Austin

Paths and Connectivity

Definition: Path

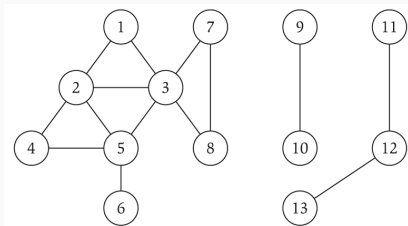
A **path** in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \dots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E .

Definition: Simple Path

A path P is **simple** if all nodes in P are distinct.

Definition: A Connected Undirected Graph

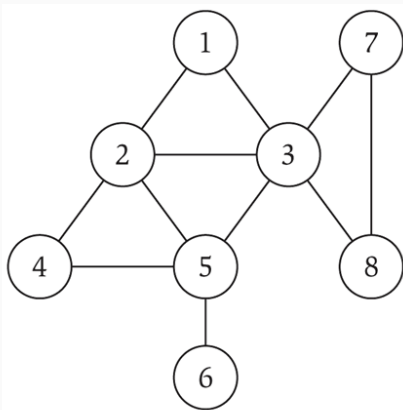
An undirected graph is **connected** if for every pair of nodes u and v , there is a path between u and v .



Cycles

Definition: Cycle

A **cycle** is a path $v_1, v_2, \dots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k - 1$ nodes are all distinct.



cycle = 1 - 2 - 4 - 5 - 3 - 1

Trees

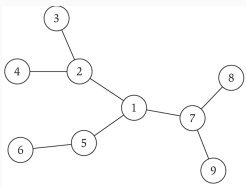
Definition: Tree

An undirected graph is a **tree** if it is connected and does not contain a cycle.

Theorem

Let G be an undirected graph on n nodes. Any two of the following statements imply the third:

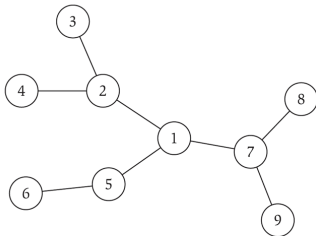
- G is connected
- G does not contain a cycle
- G has $n - 1$ edges



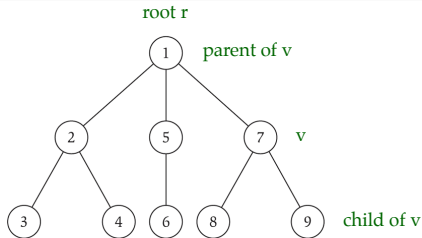
Rooted Trees

Definition: Rooted Tree

Given a tree T , choose a root node r and orient each edge away from r . This enables one to model hierarchical structure.

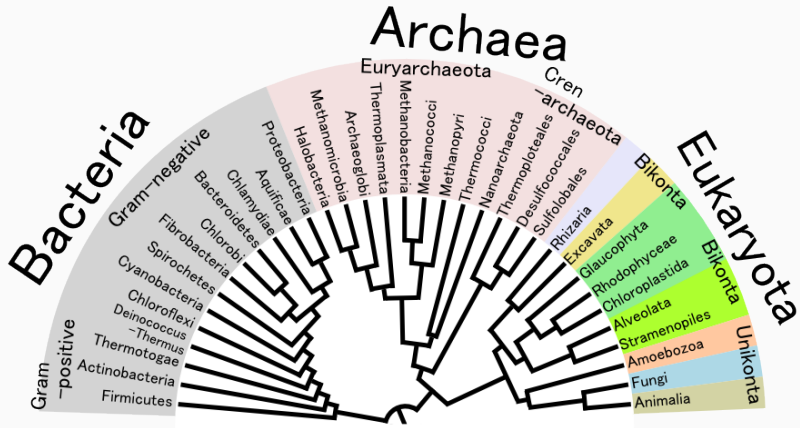


a tree



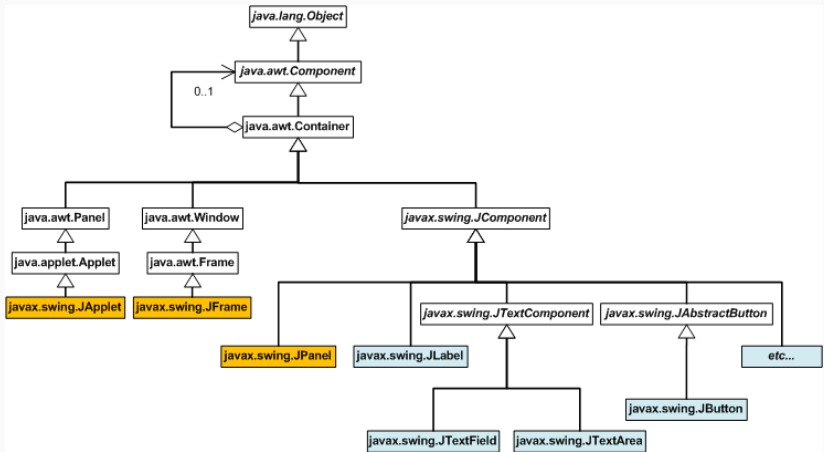
the same tree, rooted at 1

An Example Tree: Phylogeny Tree



http://en.wikipedia.org/wiki/File:Phylogenetic_Tree_of_Life.png

Another Example Tree: Object Oriented Class Architecture



<http://www.clear.rice.edu/comp310/JavaResources/GUI/>

Graph Traversal

Connectivity


$s - t$ Connectivity Problem

Given two nodes s and t , is there a path between s and t ?

$s - t$ Shortest Path Problem

Given two nodes s and t , what is the length of the shortest path between s and t ?

Applications

- Social network connections (e.g., Kevin Bacon number) 
- Maze traversal
- Fewest number of hops in a communication network

Connected Components

A Related Problem: Finding Connected Components

Find all nodes that are reachable from s .

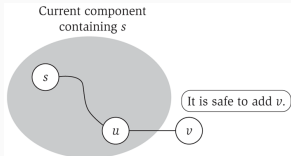
R will consist of nodes to which s has a path

Initially $R = \{s\}$

While there is an edge (u, v) where $u \in R$ and $v \notin R$

 Add v to R

Endwhile



Theorem

Upon termination, R is the connected component containing s .

Proof

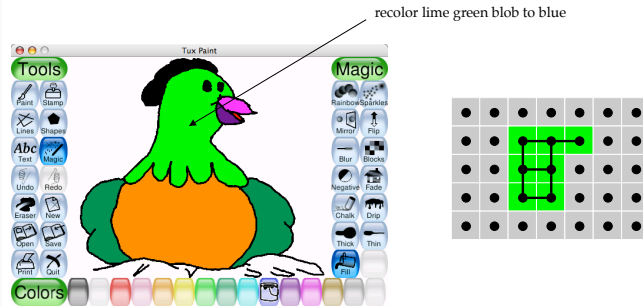
??

Connected Components: Practically

Flood Fill

Given a lime green pixel in an image, change the color of the entire blob of neighboring lime pixels to blue.

- Node: pixel
- Edge: two neighboring lime pixels
- Blob: connected component of lime pixels



Breadth First Search

The Problem

Given a graph $G = (V, E)$ and a specific source vertex s , what vertices can be reached from s ?

Not only is this problem pretty pervasive (e.g., in task scheduling), it is also a basis for other more advanced graph algorithms.

The basic idea is to systematically explore the edges of G to “discover” each node reachable from s .

- this works for both directed and undirected graphs
- the name of BFS comes from the fact that it expands the search for new nodes uniformly across the “frontier” of discovered nodes

Breadth-First Search Conceptually

Breadth First Search (BFS)

In a BFS, you can think of all nodes as being colored either white, gray, or black. Initially, all nodes are white.

- a node is “discovered” the first time the BFS encounters it; at this point BFS colors the node gray
- the complete set of gray nodes is the “frontier”
- to proceed, BFS looks at each of the gray nodes, examines each of its outgoing edges, to see if they’re connected to any white (undiscovered) nodes
 - if so, color that node gray and insert this node at the **end** of the queue of the frontier vertices
 - when we’ve examined all of a node’s outgoing edges, remove it from the frontier queue and color it black

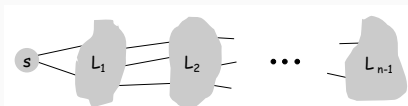
Breadth-First Search in Layers

BFS Intuition

Explore from s in all possible directions, adding nodes a “layer” at a time

BFS Algorithm

- $L_0 = \{s\}$
- $L_1 =$ all neighbors of L_0
- $L_2 =$ all nodes that do not belong to L_0 or L_1 and that have an edge to a node in L_1
- $L_{i+1} =$ all nodes that do not belong to an earlier layer and that have an edge to a node in L_i



Theorem

For each i , L_i consists of all nodes at distance exactly i from s . There is a path from s to t if and only if t appears in some layer.

Breadth First Search and Adjacency

Theorem

Let T be a breadth first search tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge of G . Then i and j differ by at most 1.

Proof

??

Breadth First Search Analysis

Assume that we use adjacency lists to store the graph, a queue to keep track of frontier nodes, and an array to keep track of which nodes were “discovered” (for each node: 0 or 1).

- $O(n)$ time to initialize the array indicating discovered or not.
- each node is discovered at most once; queue operations are $O(1)$ at most; at most $O(m)$ time is spent interacting with the queue
- each adjacency list is scanned at most once (when the node is explored); so the total time spent looking at adjacency lists is $O(2m) = O(m)$

So the total running time of breadth first search is $O(n + m)$, or linear in size to the adjacency list representation.

BFS and Shortest Paths

The level of a node in a breadth first search is the *distance* computed by the breadth first search algorithm from s to u . We define the **shortest-path distance**, $d(s, v)$ from s to v as the minimum number of edges in any path from s to v

- if there is no path from s to v , then $d(s, v) = \infty$

It is a non-trivial fact that the levels computed in breadth first search are the shortest distances from s to any node u . We'll revisit this problem in Chapter 4.

Depth-First Search

Depth First Search (DFS)

An alternative to exploring across the entire frontier at the same time is to explore a single path as far as it can go, then explore a different one.

- depth-first search explores “deeper” into the graph whenever possible
- edges are explored out of the most recently discovered vertex (v) until there are no more
- then the search backtracks, exploring other paths out of v 's parent

DFS(u):

Mark u as "Explored" and add u to R



For each edge (u, v) incident to u

 If v is not marked "Explored" then

 Recursively invoke DFS(v)

 Endif

Endfor

Theorem

Let T be a depth-first search tree, let x and y be nodes in T , and let (x, y) be an edge of G that is not an edge of T . Then one of x or y is an ancestor of the other in T .

Proof

??

Hint

In a given recursive call $DFS(u)$, all nodes marked "Explored" within this recursive call are descendants of u in T .

Comparing BFS and DFS

Comparing BFS and DFS

Similarities

- Both build the strongly connected component of G that contains s .
- Both have similar efficiency

Differences

- They explore the vertices of G in very different orders.
- They result in trees rooted at s that have very different structure (bushy vs. tall)

Questions
