

3.1 Basic Definitions and Applications

Undirected Graphs

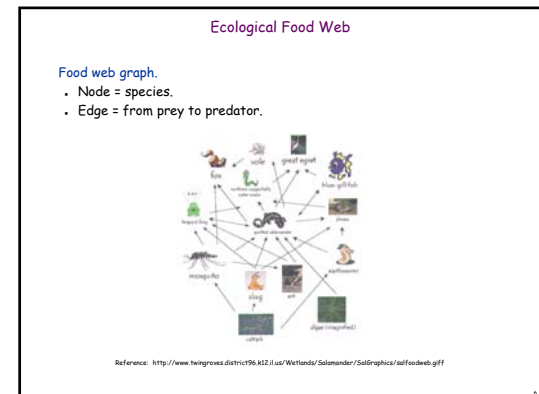
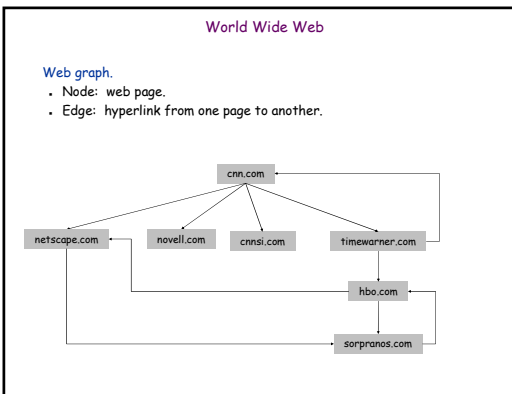
Undirected graph. $G = (V, E)$

- V = nodes.
- E = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: $n = |V|$, $m = |E|$.

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

Some Graph Applications

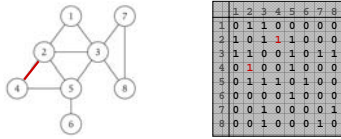
Graph	Nodes	Edges
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires



Graph Representation: Adjacency Matrix

Adjacency matrix. n -by- n matrix with $A_{uv} = 1$ if (u, v) is an edge.

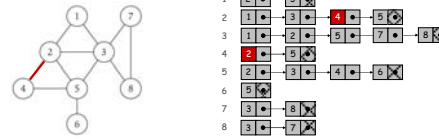
- Two representations of each edge.
- Space proportional to n^2 .
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.



Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.

- Two representations of each edge.
- Space proportional to $m + n$.
- Checking if (u, v) is an edge takes $O(\text{deg}(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.

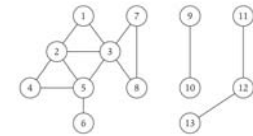


Paths and Connectivity

Def. 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 .

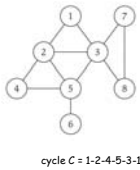
Def. A path is **simple** if all nodes are distinct.

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



Cycles

Def. 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.

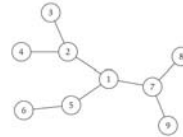


Trees

Def. 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.

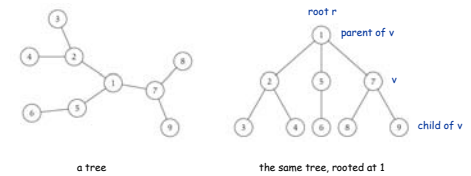
- G is connected.
- G does not contain a cycle.
- G has $n-1$ edges.



Rooted Trees

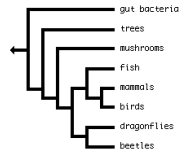
Rooted tree. Given a tree T , choose a root node r and orient each edge away from r .

Importance. Models hierarchical structure.



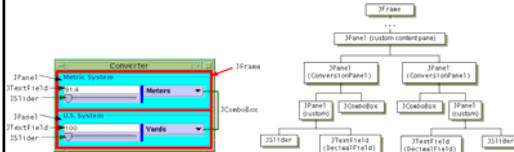
Phylogeny Trees

Phylogeny trees. Describe evolutionary history of species.



GUI Containment Hierarchy

GUI containment hierarchy. Describe organization of GUI widgets.



Reference: <http://java.sun.com/docs/books/tutorial/uiswing/overview/antony.html>

3.2 Graph Traversal

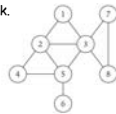
Connectivity

s-t connectivity problem. Given two node s and t, is there a path between s and t?

s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

Applications.

- Friendster.
- Maze traversal.
- Kevin Bacon number.
- Fewest number of hops in a communication network.

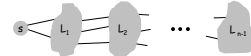


Breadth First Search

BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.

BFS algorithm.

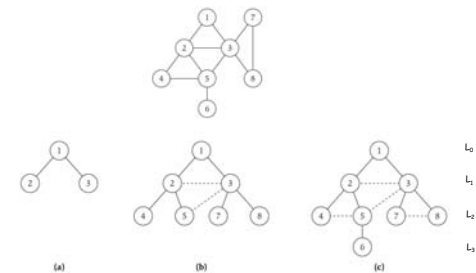
- $L_0 = \{s\}$.
- L_1 = all neighbors of L_0 .
- L_2 = all nodes that do not belong to L_0 or L_1 , and that have an edge to a node in L_1 .
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i .



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

Breadth First Search

Property. Let T be a BFS tree of $G = (V, E)$, and let (x, y) be an edge of G. Then the level of x and y differ by at most 1.



Breadth First Search: Analysis

Theorem. The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

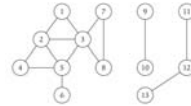
Pf.

- Easy to prove $O(n^2)$ running time:
 - at most n lists $L[i]$
 - each node occurs on at most one list; for loop runs $\leq n$ times
 - when we consider node u , there are $\leq n$ incident edges (u, v) , and we spend $O(1)$ processing each edge
- Actually runs in $O(m + n)$ time:
 - when we consider node u , there are $\text{deg}(u)$ incident edges (u, v)
 - total time processing edges is $\sum_u \text{deg}(u) = 2m$

each edge (u, v) is counted exactly twice in sum: once in $\text{deg}(u)$ and once in $\text{deg}(v)$

Connected Component

Connected component. Find all nodes reachable from s .



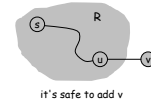
Connected component containing node 1 = { 1, 2, 3, 4, 5, 6, 7, 8 }.

Connected Component

Connected component. Find all nodes 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 ∈ R and v ∉ R
  Add v to R
Endwhile
    
```



- Theorem.** Upon termination, R is the connected component containing s .
- BFS = explore in order of distance from s .
 - DFS = explore in a different way.

Flood Fill

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

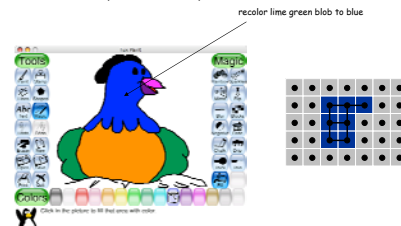
- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.



Flood Fill

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

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



Implementing Breadth-first Search

- Maintain an array **Discovered** of length n .
- Set **Discovered**[v] = true when the search sees v .
- Constructs layers of nodes L_1, L_2, \dots in lists $L[i]$.

```

Set Discovered[s]=true and Discovered[v]=false for all other v
Initialize L[0] to consist a single element s
Set the layer counter i=0
Set the current BFS tree T=∅
While L[i] is not empty
  Initialize an empty list L[i+1]
  For each node u in L[i]
    Consider each edge (u,v) incident to u
    If Discovered[v]=false then
      Set Discovered[v]=true
      Add edge (u,v) to the tree T
      Add v to the list L[i+1]
    Endif
  Endfor
  Increment i by one
Endwhile
    
```

Implementing Breadth-first Search

- This implementation runs in $O(m+n)$ -time if the graph is given by the adjacency list representation.

Proof:

- Easy to bound by $O(n^2)$
 - Each node occurs on at most one list.
 - For loop runs at most n times.
- Consider a node u : There can be at most n edges incident to u .
 - Total time spent on one iteration of for is at most $O(n)$.
- To get the improved time $O(n+m)$:
 - Let n_u denote the degree of node u
 - Time spent in for loop considering a node is $O(n_u)$
 - The total over all nodes $O(\sum_u n_u) = O(m)$
 - We need additional time to set up lists and manage the array Discovered.

24

Implementing Breadth-first Search

- We can implement the algorithm using a single list that is maintained as a queue.
- Each time a node is discovered it is added to the end of queue.
- The algorithm always processes the edges of the node that is first in the queue.
- All nodes in layer L_i appear in the queue ahead of all nodes in layer L_{i+1}

25

Implementing Depth-first Search

- Maintain the nodes to be processed in a stack.
- When DFS explores a node u , it scans the neighbors of u until it finds the first not-yet-explored node v .
- Use an array Explored.
- Set Explored[v]=true when the v 's incident edges are scanned.

```
Initialize S to be a stack with one element s
While S is not empty
  Take a node u from S
  If Explored[u]=false then
    Set Explored[u]=true
    For each edge (u,v) incident to u
      Add v to the stack S
    Enfor
  Endif
Endwhile
```

27

Implementing Depth-first Search

- The main step is to add and delete nodes to and from the stack.
- To bound the running time we need to bound the number of these operations.
- So, count the number of nodes added to stack.

How many elements get added to S?

- Let n_v denote the degree of node v .
- v will be added to the stack every time one of its adjacent nodes is explored.
- So the total number of nodes added is at most $2m$.

This implementation of DFS runs in time $O(m+n)$ if the graph is given by the adjacency list representation.

28

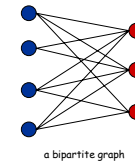
3.4 Testing Bipartiteness

Bipartite Graphs

Def. An undirected graph $G = (V, E)$ is **bipartite** if the nodes can be colored red or blue such that every edge has one red and one blue end.

Applications.

- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.

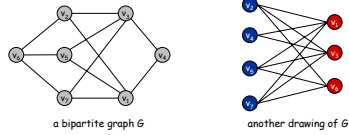


30

Testing Bipartiteness

Testing bipartiteness. Given a graph G , is it bipartite?

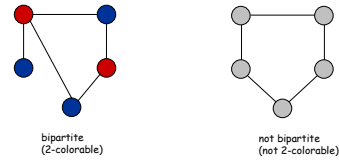
- Many graph problems become:
 - easier if the underlying graph is bipartite (matching)
 - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.



An Obstruction to Bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

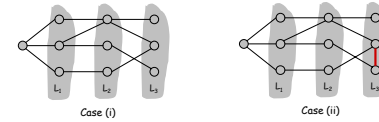
Pf. Not possible to 2-color the odd cycle, let alone G .



Bipartite Graphs

Lemma. Let G be a connected graph, and let L_0, \dots, L_k be the layers produced by BFS starting at node s . Exactly one of the following holds.

- No edge of G joins two nodes of the same layer, and G is bipartite.
- An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



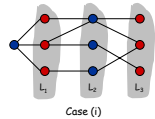
Bipartite Graphs

Lemma. Let G be a connected graph, and let L_0, \dots, L_k be the layers produced by BFS starting at node s . Exactly one of the following holds.

- No edge of G joins two nodes of the same layer, and G is bipartite.
- An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Pf. (i)

- Suppose no edge joins two nodes in the same layer.
- This implies all edges join nodes on different levels.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.



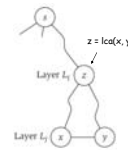
Bipartite Graphs

Lemma. Let G be a connected graph, and let L_0, \dots, L_k be the layers produced by BFS starting at node s . Exactly one of the following holds.

- No edge of G joins two nodes of the same layer, and G is bipartite.
- An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Pf. (ii)

- Suppose (x, y) is an edge with x, y in same level L_j .
- Let $z = \text{lca}(x, y)$ = lowest common ancestor.
- Let L_i be level containing z .
- Consider cycle that takes edge from x to y , then path from y to z , then path from z to x .
- Its length is $1 + \underbrace{(j-i)}_{\text{path from } y \text{ to } z} + \underbrace{(j-i)}_{\text{path from } z \text{ to } x}$, which is odd.

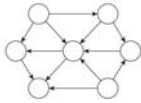


3.5 Connectivity in Directed Graphs

Directed Graphs

Directed graph. $G = (V, E)$

- Edge (u, v) goes from node u to node v .



Ex. Web graph - hyperlink points from one web page to another.

- Directedness of graph is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.

37

Graph Search

Directed reachability. Given a node s , find all nodes reachable from s .

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

Graph search. BFS extends naturally to directed graphs.

Web crawler. Start from web page s . Find all web pages linked from s , either directly or indirectly.

38

Strong Connectivity

Def. Node u and v are **mutually reachable** if there is a path from u to v and also a path from v to u .

Def. A graph is **strongly connected** if every pair of nodes is mutually reachable.

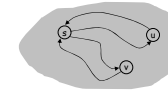
Lemma. Let s be any node. G is strongly connected iff every node is reachable from s , and s is reachable from every node.

Pf. \Rightarrow Follows from definition.

Pf. \Leftarrow Path from u to v : concatenate u - s path with s - v path.

Path from v to u : concatenate v - s path with s - u path. •

ok if paths overlap



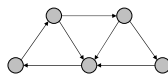
39

Strong Connectivity: Algorithm

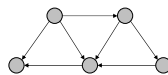
Theorem. Can determine if G is strongly connected in $O(m + n)$ time.

Pf.

- Pick any node s .
- Run BFS from s in G .
- Run BFS from s in G^{rev} . reverse orientation of every edge in G
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. •



strongly connected



not strongly connected

40

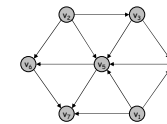
3.6 DAGs and Topological Ordering

Directed Acyclic Graphs

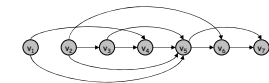
Def. An **DAG** is a directed graph that contains no directed cycles.

Ex. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j .

Def. A **topological order** of a directed graph $G = (V, E)$ is an ordering of its nodes as v_1, v_2, \dots, v_n so that for every edge (v_i, v_j) we have $i < j$.



a DAG



a topological ordering

42

Precedence Constraints

Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j .

Applications.

- Course prerequisite graph: course v_i must be taken before v_j .
- Compilation: module v_i must be compiled before v_j . Pipeline of computing jobs: output of job v_i needed to determine input of job v_j .

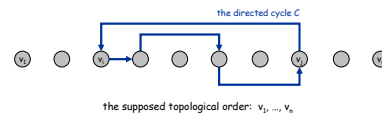
43

Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG.

Pf. (by contradiction)

- Suppose that G has a topological order v_1, \dots, v_n and that G also has a directed cycle C . Let's see what happens.
- Let v_i be the lowest-indexed node in C , and let v_j be the node just before v_i ; thus (v_j, v_i) is an edge.
- By our choice of i , we have $i < j$.
- On the other hand, since (v_j, v_i) is an edge and v_1, \dots, v_n is a topological order, we must have $j < i$, a contradiction. •



44

Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG.

Q. Does every DAG have a topological ordering?

Q. If so, how do we compute one?

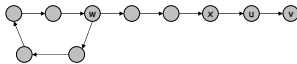
45

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)

- Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node v , and begin following edges backward from v . Since v has at least one incoming edge (u, v) we can walk backward to u .
- Then, since u has at least one incoming edge (x, u) , we can walk backward to x .
- Repeat until we visit a node, say w , twice.
- Let C denote the sequence of nodes encountered between successive visits to w . C is a cycle. •



46

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a topological ordering.

Pf. (by induction on n)

- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node v with no incoming edges.
- $G - \{v\}$ is a DAG, since deleting v cannot create cycles.
- By inductive hypothesis, $G - \{v\}$ has a topological ordering.
- Place v first in topological ordering; then append nodes of $G - \{v\}$
- in topological order. This is valid since v has no incoming edges. •

To compute a topological ordering of G :
 Find a node v with no incoming edges and order it first
 Delete v from G
 Recursively compute a topological ordering of $G - \{v\}$
 and append this order after v



47

Topological Sorting Algorithm: Running Time

Theorem. Algorithm finds a topological order in $O(m + n)$ time.

Pf.

- Maintain the following information:
 - $\text{count}[w]$ = remaining number of incoming edges
 - S = set of remaining nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete v
 - remove v from S
 - decrement $\text{count}[w]$ for all edges from v to w , and add w to S if $\text{count}[w]$ hits 0
 - this is $O(1)$ per edge. •

48