CSE 421: Intro Algorithms

Fall 2011
Graphs and Graph Algorithms
Slides by Larry Ruzzo

Goals
Graphs: defns, examples, utility, terminology
Representation: input, internal
Traversal: Breadth- & Depth-first search
Three Algorithms:
  Connected components
  Bipartiteness
  Topological sort

Objects & Relationships
The Kevin Bacon Game:
Obj: Actors
Rel: Two are related if they've been in a movie together
Exam Scheduling:
Obj: Classes
Rel: Two are related if they have students in common
Traveling Salesperson Problem:
Obj: Cities
Rel: Two are related if can travel directly between them

Graphs
An extremely important formalism for representing (binary) relationships
Objects: "vertices," aka "nodes"
Relationships between pairs: "edges," aka "arcs"
Formally, a graph G = (V, E) is a pair of sets,
V the vertices and E the edges
Undirected Graph $G = (V, E)$

Graphs don’t live in Flatland
Geometrical drawing is mentally convenient, but mathematically irrelevant: 4 drawings, 1 graph.

Directed Graph $G = (V, E)$ 
Directed Graph $G = (V,E)$

Directed Graph $G = (V,E)$

Directed Graph $G = (V,E)$

Directed Graph $G = (V,E)$

Directed Graph $G = (V,E)$

Directed Graph $G = (V,E)$

Directed Graph $G = (V,E)$

Specifying undirected graphs as input

What are the vertices?
Explicitly list them:
{"A", "7", "3", "4"}

What are the edges?
Either, set of edges
{{A,3}, {7,4}, {4,3}, {4,A}}
Or, (symmetric) adjacency matrix:

```
A  7  3  4
A  0  0  1  1
7  0  0  0  1
3  1  0  0  1
4  1  1  1  0
```

Specifying directed graphs as input

What are the vertices?
Explicitly list them:
{"A", "7", "3", "4"}

What are the edges?
Either, set of directed edges:
{(A,4), (4,7), (4,3), (4,A)}
Or, (nonsymmetric) adjacency matrix:

```
A  7  3  4
A  0  0  1  1
7  0  0  0  0
3  0  0  0  0
4  1  1  1  0
```
Let $G$ be an undirected graph with $n$ vertices and $m$ edges. How are $n$ and $m$ related?

Since every edge connects two different vertices (no loops), and no two edges connect the same two vertices (no multi-edges), it must be true that:

$$0 \leq m \leq \frac{n(n-1)}{2} = O(n^2)$$

A graph is called sparse if $m \ll n^2$, otherwise it is dense.

Sparse graphs are common in practice. E.g., all planar graphs are sparse ($m \leq 3n-6$, for $n \geq 3$).

Q: which is a better run time, $O(n+m)$ or $O(n^2)$?

A: $O(n+m) = O(n^2)$, but $n+m$ usually way better!

Representing Graph $G = (V,E)$

- Vertex set $V = \{v_1, \ldots, v_n\}$
- Adjacency Matrix $A$
- $A[i,j] = 1$ if $(v_i, v_j) \in E$
- Space is $n^2$ bits

Advantages:
- $O(1)$ test for presence or absence of edges.
- Efficient for sparse graphs, both in storage and access.

Disadvantages:
- $m \ll n^2$.

Representing Graph $G=(V,E)$

- $n$ vertices, $m$ edges
- Adjacency List:
  - $O(n+m)$ words
- Advantages:
  - Compact for sparse graphs
  - Easily see all edges
- Disadvantages:
  - More complex data structure
  - No $O(1)$ edge test

Representing Graph $G=(V,E)$

- $n$ vertices, $m$ edges
- Adjacency List:
  - $O(n+m)$ words
- Back- and cross pointers more work to build, but allow easier traversal and deletion of edges, if needed, (don’t bother if not)
Graph Traversal
Learn the basic structure of a graph
“Walk,” via edges, from a fixed starting vertex
s to all vertices reachable from s

Being orderly helps. Two common ways:
Breadth-First Search: order the nodes in
successive layers based on distance from s
Depth-First Search: more natural approach for
exploring a maze; many efficient algs build on it.

Breadth-First Search
Completely explore the vertices in order of
their distance from s

Naturally implemented using a queue

Graph Traversal: Implementation
Learn the basic structure of a graph
“Walk,” via edges, from a fixed starting vertex
s to all vertices reachable from s

Three states of vertices
undiscovered
discovered
fully-explored

BFS(s) Implementation
Global initialization: mark all vertices “undiscovered”
BFS(s)
mark s “discovered”
queue = { s }
while queue not empty
u = remove_first(queue)
for each edge (u,x)
if (x is undiscovered)
mark x discovered
append x on queue
mark u fully explored

Exercise: modify
code to number
vertices & compute
level numbers
BFS(v)

Queue: 3 4

Queue: 4 5 6 7

Queue: 5 6 7 8 9

Queue: 8 9 10 11

Queue: 10 11 12 13

Queue:
BFS: Analysis, I

Global initialization: mark all vertices "undiscovered"
mark s "discovered"

Queue = {s}

while queue not empty
    u = remove_first(queue)
    for each edge {u, x}
        if (x is undiscovered)
            mark x discovered
            append x on queue
    mark u fully explored

Simple analysis:
2 nested loops.
Get worst-case number of iterations of each; multiply.

BFS: Analysis, II

Above analysis correct, but pessimistic (can’t have \( \Omega(n) \) edges incident to each of \( \Omega(n) \) distinct "u" vertices if G is sparse). Alt, more global analysis:

Each edge is explored once from each end-point, so total runtime of inner loop is \( O(m) \).

Total \( O(n+m) \), \( n = \# \) nodes, \( m = \# \) edges

Properties of (Undirected) BFS(v)

BFS(v) visits x if and only if there is a path in G from v to x.
Edges into then undiscovered vertices define a tree – the "breadth first spanning tree" of G.

Level i in this tree are exactly those vertices u such that the shortest path (in G, not just the tree) from the root v is of length i.

All non-tree edges join vertices on the same or adjacent levels

BFS Application: Shortest Paths

Tree (solid edges) gives shortest paths from start vertex
can label by distances from start
all edges connect same/adjacent levels

Exercise: extend algorithm and analysis to non-connected graphs

BFS Application: Shortest Paths

Tree (solid edges) gives shortest paths from start vertex
can label by distances from start
all edges connect same/adjacent levels
**Tree (solid edges) gives shortest paths from start vertex**

**BFS Application: Shortest Paths**

- Can label by distances from start
- All edges connect same/adjacent levels

**Why fuss about trees?**

- Trees are simpler than graphs
- Ditto for algorithms on trees vs algs on graphs

So, this is often a good way to approach a graph problem: find a "nice" tree in the graph, i.e., one such that non-tree edges have some simplifying structure.

E.g., BFS finds a tree s.t. level-jumps are minimized.

DFS (below) finds a different tree, but it also has interesting structure...

**Graph Search Application: Connected Components**

Want to answer questions of the form: given vertices u and v, is there a path from u to v?

Set up one-time data structure to answer such questions efficiently.

**Graph Search Application: Connected Components**

Want to answer questions of the form: given vertices u and v, is there a path from u to v?

Idea: create array A such that:

- A[u] = smallest numbered vertex that is connected to u.

Question reduces to whether A[u] = A[v]?

**Graph Search Application: Connected Components**

- Initial state: all v undiscovered
- for v = 1 to n do
  - if state(v) != fully-explored then
    - BFS(v): setting A[u] ← v for each u found
    - (and marking u discovered/fully-explored)
  - endif
- endfor

Total cost: O(n+m)

each edge is touched a constant number of times (twice)

works also with DFS

**3.4 Testing Bipartiteness**
Bipartite Graphs

Def. An undirected graph $G = (V, E)$ is bipartite (2-colorable) if the nodes can be colored red or blue such that no edge has both ends the same color.

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

A bipartite graph "bi-partite" means "two parts." An equivalent definition: $G$ is bipartite if you can partition the node set into 2 parts (red, blue) so that all edges join nodes in different parts.

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. Impossible to 2-color the odd cycle, let alone $G$.
Obstruction to Bipartiteness

Cor: A graph $G$ is bipartite iff it contains no odd length cycle.

NB: the proof is algorithmic: it finds a coloring or odd cycle.

3.6 DAGs and Topological Ordering

Directed Acyclic Graphs

Def. A DAG is a directed acyclic graph, i.e., one 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 $v_1, v_2, \ldots, v_n$ so that for every edge $(v_i, v_j)$ we have $i < j$.

E.g., edge $(v_2, v_3)$; finish $v_2$ before starting $v_3$.

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 $v_1, \ldots, v_n$ and that $G$ also has a directed cycle $C$. 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 $(v_i, v_j)$ is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction.

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?
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 \( C \) be the sequence of nodes encountered between successive visits to \( w \). \( C \) is a cycle.

Why must this happen?

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.

\[ \square \]

Topological Ordering Algorithm: Example

To compute a topological ordering of \( G \):

1. Find a node \( v \) with no incoming edges and order it first.
2. Delete \( v \) from \( G \).
3. Recursively compute a topological ordering of \( G - \{v\} \) and append this order after \( v \).

Topological order: \( v_1, v_2, v_3 \)
Topological Ordering Algorithm: Example

Topological order: v₁, v₂, v₃, v₄, v₅, v₆

Topological Sorting Algorithm

Maintain the following:
- count[w] = (remaining) number of incoming edges to node w
- S = set of (remaining) nodes with no incoming edges

Initialization:
- count[w] = 0 for all w
- count[w]++ for all edges (v,w)
- S = S ∪ {w} for all w with count[w]=0

Main loop:
- while S not empty
  - remove some v from S
  - make v next in topo order
  - for all edges from v to some w
    - decrement count[w]
  - add w to S if count[w] hits 0

Correctness: clear, I hope
Time: O(m + n) (assuming edge-list representation of graph)

Depth-First Search

Follow the first path you find as far as you can
Back up to last unexplored edge when you reach a dead end, then go as far you can

Naturally implemented using recursive calls or a stack
Non-tree edges
All non-tree edges join a vertex and one of its descendents/ancestors in the DFS tree
No cross edges!

DFS(v) – Recursive version
Global Initialization:
for all nodes v, v.dfs = -1 // mark v "undiscovered"
dcounter = 0
DFS(v)
v.dfs = dcounter++ // v "discovered", number it
for each edge (v,x)
if (x.dfs = -1) // tree edge (x previously undiscovered)
    DFS(x)
else ... // code for back-, fwd-, parent, edges, if needed
mark v "completed," if needed.

DFS(A) - Explicit stack
Global Initialization: mark all vertices "undiscovered"
DFS(v)
mark v "discovered"
push (v,1) onto empty stack
while stack not empty
    (u,i) = pop(stack)
    for ( ; i ≤ # of neighbors of u; i++)
        x = i
            edge on u's edge list
    if (x is undiscovered)
        mark x "discovered"
push (u,i+1) // save info to resume with u's next edge,
    if (x = u)
        i = 1 // (starting with its first edge)
mark u completed
Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- **red**: undiscovered
- **green**: discovered
- **blue**: fully-explored

Call Stack:

DFS(A)

1. **A**
   - **B**
   - **J**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**
   - **N**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**
   - **N**
   - **O**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**
   - **N**
   - **O**
   - **P**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**
   - **N**
   - **O**
   - **P**
   - **Q**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**
   - **N**
   - **O**
   - **P**
   - **Q**
   - **R**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**
   - **N**
   - **O**
   - **P**
   - **Q**
   - **R**
   - **S**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**
   - **N**
   - **O**
   - **P**
   - **Q**
   - **R**
   - **S**
   - **T**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**
   - **N**
   - **O**
   - **P**
   - **Q**
   - **R**
   - **S**
   - **T**
   - **U**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**
   - **N**
   - **O**
   - **P**
   - **Q**
   - **R**
   - **S**
   - **T**
   - **U**
   - **V**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**
   - **N**
   - **O**
   - **P**
   - **Q**
   - **R**
   - **S**
   - **T**
   - **U**
   - **V**
   - **W**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**
   - **N**
   - **O**
   - **P**
   - **Q**
   - **R**
   - **S**
   - **T**
   - **U**
   - **V**
   - **W**
   - **X**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**
   - **N**
   - **O**
   - **P**
   - **Q**
   - **R**
   - **S**
   - **T**
   - **U**
   - **V**
   - **W**
   - **X**
   - **Y**

DFS(A)

1. **A**
   - **B**
   - **C**
   - **D**
   - **E**
   - **F**
   - **G**
   - **H**
   - **I**
   - **J**
   - **K**
   - **L**
   - **M**
   - **N**
   - **O**
   - **P**
   - **Q**
   - **R**
   - **S**
   - **T**
   - **U**
   - **V**
   - **W**
   - **X**
   - **Y**
   - **Z**
Suppose edge lists at each vertex are sorted alphabetically.
Suppose edge lists at each vertex are sorted alphabetically.
Suppose edge lists at each vertex are sorted alphabetically.
Suppose edge lists at each vertex are sorted alphabetically.
Properties of (Undirected) DFS(v)

Like BFS(v):
- DFS(v) visits x if and only if there is a path in G from v to x (through previously unvisited vertices)
- Edges into then-undiscovered vertices define a tree – the "depth first spanning tree" of G

Unlike the BFS tree:
- the DF spanning tree isn't minimum depth
- its levels don't reflect min distance from the root
- non-tree edges NEVER join vertices on the same or adjacent levels

BUT...

Non-tree edges

All non-tree edges join a vertex and one of its descendents/ancestors in the DFS tree

No cross edges!

A simple problem on trees

Given: tree T, a value L(v) defined for every vertex v in T
Goal: find M(v), the min value of L(v) anywhere in the subtree rooted at v (including v itself).

How?
A simple problem on trees

Given: tree T, a value $L(v)$ defined for every vertex $v$ in $T$
Goal: find $M(v)$, the min value of $L(v)$ anywhere in the subtree rooted at $v$
(including $v$ itself).

How? Depth first search, using:

$$
M(v) = \begin{cases} 
L(v) & \text{if } v \text{ is a leaf} \\
\min(L(v), \min_{w \text{ a child of } v} M(w)) & \text{otherwise}
\end{cases}
$$

Application: Articulation Points

A node in an undirected graph is an **articulation point** iff removing it disconnects the graph.

articulation points represent vulnerabilities in a network – single points whose failure would split the network into 2 or more disconnected components.

Articulation Points

articulation point
iff its removal disconnects the graph

Brainstorming

draw a graph, ~ 10 nodes, A-J
redraw as via DFS, starting at "E"
add dsdfs & tree/back edges (solid/dashed)
find cycles
give alg to find cycles via dfs; does G have any?
find articulation points
what do cycles have to do with articulation points?
alg to find articulation points via DFS???
Simple Case: Artic. Pts in a tree
Leaves – never articulation points
Internal nodes – always articulation points
Root – articulation point if and only if two or more children
Non-tree: extra edges remove some articulation points (which ones?)

Articulation Points from DFS
Root node is an articulation point
iff it has more than one child
Leaf is never an articulation point
non-leaf, non-root node u is an articulation point
3 some child y of u s.t. no non-tree edge goes above u from y or below
If removal of u does NOT separate x, there must be an exit from x's subtree. How?
Via back edge.

Articulation Points:
the "LOW" function
Definition: $LOW(v)$ is the lowest dfs# of any vertex that is either in the dfs subtree rooted at $v$
(including v itself) or connected to a vertex in that subtree by a back edge.

DFS(v) for Finding Articulation Points
Global initialization: v.dfs# = -1 for all v.
DFS(x)
v.dfs# = dfscounter++
v.low = v.dfs#
// initialization
for each edge {x,y}
if (x.dfs# == -1) // x is undiscovered
  DFS(x)
  v.low = min(v.low, x.low)
if (x.low >= v.dfs#)
  print "v is an articulation point, separating x"
else if (x is non-v's parent)
  v.low = min(v.low, x.dfs#)
Equiv: "If (x,y) is a back edge"
Why?
Summary
Graphs – abstract relationships among pairs of objects
Terminology – node/vertex/vertices, edges, paths, multi-edges, self-loops, connected
Representation – edge list, adjacency matrix
Nodes vs Edges – $m = O(n^2)$, often less
BFS – Layers, queue, shortest paths, all edges go to same or adjacent layer
DFS – recursion/stack; all edges ancestor/descendant
Algorithms – connected components, bipartiteness, topological sort, articulation points
Articulation Points from DFS

- Every interior vertex of a tree is an articulation point.
- Non-tree edges eliminate articulation points.
- Leaves are never articulation points.
- Root node is an articulation point iff it has more than one child.

DFS Application: Articulation Points

- Non-tree edge goes above $u$ from a sub-tree below some child of $u$. 

Articulation Points: Some Subtleties

- 4, 5, 6 should be eliminated, yet are unmatched.
- 3, 8, 12 sub-tree at 4, 9, 11 sub-tree at 13.

DFS Vertex Numbering

If $u$ is an ancestor of $v$ in the DFS tree, then $dfs#(u) < dfs#(v)$. 