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 \( G = (V,E) \)
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 $G = (V,E)$

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:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>7</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
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), (A,3)}
Or, (nonsymmetric) adjacency matrix:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>7</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

# Vertices vs # Edges

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:

\[ 0 \leq m \leq n(n-1)/2 = O(n^2) \]

More Cool Graph Lingo

A graph is called \textit{sparse} if \( m \ll n^2 \), otherwise it is \textit{dense}.
Boundary is somewhat fuzzy; \( O(n) \) edges is certainly sparse, \( \Omega(n^2) \) edges is dense.
Sparse graphs are common in practice
E.g., all planar graphs are sparse \((m \leq 3n-6, \text{ for } n \geq 3)\)
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.
- **Disadvantages:** inefficient for sparse graphs, both in storage and access

\[
\begin{array}{cccc}
4 & 7 & 3 & 4 \\
7 & 9 & 0 & 0 \ \\
3 & 1 & 0 & 1 \\
4 & 1 & 1 & 0 \\
\end{array}
\]

\( m \ll n^2 \)

---

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

\[
\begin{array}{c}
V_1 \\
V_2 \\
V_3 \\
V_7 \\
\end{array}
\]

---

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"

\[
\text{BFS}(s) \\
\begin{array}{l}
\text{mark } s \text{ "discovered"} \\
\text{queue } = \{ s \} \\
\text{while queue not empty} \\
\text{u = remove\_first(queue)} \\
\text{for each edge } \{u,x\} \\
\quad \text{if } x \text{ is undiscovered} \\
\quad \text{mark } x \text{ discovered} \\
\quad \text{append } x \text{ on queue} \\
\text{mark u fully explored}
\end{array}
\]

Exercise: modify code to number vertices & compute level numbers

BFS(v)
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

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

\[ O(n) + O(1) + O(n) \times O(n) = O(n^2) \]
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
BFS Application: Shortest Paths

Tree (solid edges)
gives shortest paths from start vertex

1 0
2
3 4

can label by distances from start all edges connect same/adjacent levels

BFS Application: Shortest Paths

Tree (solid edges)
gives shortest paths from start vertex

1 0
2
3 4

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…
**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

**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.

**Exercise:** modify code to number vertices & compute level numbers

Label edges as tree edges or non-tree edges

**Graph Search Application: Connected Components**

initial state: all v undiscovered

for v = 1 to n do
  - if state(v) != fully-explored then
    - BFS(v)
  - endif
endfor

Total cost: \(O(n+m)\)
  - each edge is touched a constant number of times (twice)
  - works also with DFS

**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

\[ A[u] = \text{smallest numbered vertex that is connected to } u. \]

**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 \( A[u] = A[v] \)?

Q: Why not create 2-d array \( \text{Path}[u,v] \)?

**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 \( A[u] \leftarrow v \) 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

"bi-partite" means "two parts." An equivalent definition: \( G \) is bipartite if you can partition the node set into 2 parts (say, blue/red or left/right) so that all edges join nodes in different parts/no edge has both ends in the same part.
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 $G$.

---

Bipartite Graphs

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

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) 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. By previous lemma, all edges join nodes on adjacent levels.

Bipartition:
- red = nodes on odd levels,
- blue = nodes on even levels.
**Bipartite Graphs**

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

(i) No edge of G joins two nodes of the same layer, and G is bipartite.
(ii) 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 & x, y in same level $L_j$.
Let z = their lowest common ancestor in BFS tree.
Let $L_i$ be level containing z.
Consider cycle that takes edge from x to y, then tree from y to z, then tree from z to x.
Its length is $1 + (j-i) + (j-i)$, which is odd.

**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.

**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, v)
        if (v is undiscovered)
            mark v discovered
            append v on queue
mark u fully explored

**Exercise:** modify code to determine if the graph is bipartite

**3.6 DAGs and Topological Ordering**
Precedence Constraints

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

Applications

Course prerequisites: course \(v_i\) must be taken before \(v_j\).
Compilation: must compile module \(v_i\) before \(v_j\).
Computing workflow: output of job \(v_i\) is input to job \(v_j\).
Manufacturing or assembly: sand it before you paint it…
Spreadsheet evaluation order: if \(A7 = A6 + A5 + A4\), evaluate them first.

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 \((v_i, v_j)\), finish \(v_i\) before starting \(v_j\).

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 \(v_i, \ldots, v_n\) 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.

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 \(v_i, \ldots, v_n\) 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.

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.

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.
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2$

Topological order: $v_1, v_2, v_3$
Topological Ordering Algorithm: Example

Topological order: \( v_1, v_2, v_3, v_4 \)

Topological Ordering Algorithm: Example

Topological order: \( v_1, v_2, v_3, v_4, v_5, v_6 \)

Topological Ordering Algorithm: Example

Topological order: \( v_1, v_2, v_3, v_4, v_5, v_6, v_7 \)
Topological Sorting Algorithm

Linear time implementation!

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

DFS(v) – Recursive version

Global Initialization:
- for all nodes `v`, `v.dfs# = -1` // mark `v` "undiscovered"
- `dfscounter = 0`

DFS(v)
- `v.dfs# = dfscounter++` // `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`
Why fuss about trees (again)?

BFS tree ≠ DFS tree, but, as with BFS, DFS has found a tree in the graph s.t. non-tree edges are "simple" – only descendant/ancestor.
Suppose edge lists at each vertex are sorted alphabetically.
Suppose edge lists at each vertex are sorted alphabetically.

Call Stack:
- (Edge list)
  - A (G, J)
  - B (G, J)
  - C (G, J, G, H)
  - D (G, F, F, F)
  - E (G, F, F)
  - F (G, E, G, F)
  - G (G, F, F)

Color code:
- undiscovered
- discovered
- fully-explored
Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)

DFS(A)
- A,1
- B,2
- C,3
- D,4
- E,5
- F,6
- G,7
- H,8
- I
- J
- K
- L
- M
Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- **undiscovered**
- **discovered**
- **fully-explored**

Call Stack:
(Edge list)

A (F, J)
B (G, J)
C (B, D, G, H)
H (F, J)
I (F)

A, 1
B, 2
J
I, 9
H, 8
C, 3
G, 7
F, 6
D, 4
E, 5
K
L
M

A, 1
B, 2
J
I, 9
H, 8
C, 3
G, 7
F, 6
D, 4
E, 5
K
L
M

A, 1
B, 2
J, 10
I, 9
H, 8
C, 3
G, 7
F, 6
D, 4
E, 5
K
L
M

A, 1
B, 2
J, 10
I, 9
H, 8
C, 3
G, 7
F, 6
D, 4
E, 5
K
L
M
Suppose edge lists at each vertex are sorted alphabetically.
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Color code:
- **undiscovered**
- **discovered**
- **fully-explored**

Call Stack:
- (Edge list)

Graphical representation showing a tree structure with vertices labeled A to M, and edges connecting them. The vertices are colored to indicate their status: undiscovered, discovered, or fully-explored.
Suppose edge lists at each vertex are sorted alphabetically.

**Color code:**
- undiscovered
- discovered
- fully-explored

**Call Stack:**
- (Edge list)

DFS(A)

1. A, 1
2. B, 2
3. J, 10
4. I, 9
5. H, 8
6. C, 3
7. G, 7
8. F, 6
9. D, 4
10. E, 5

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!

Why fuss about trees (again)?

As with BFS, DFS has found a tree in the graph s.t. non-tree edges are "simple"--only descendant/ancestor

DFS(v) – Recursive version

Global Initialization:
- for all nodes v, v.dfs# = -1 // mark v "undiscovered"
- dfscounter = 0

DFS(v)
- v.dfs# = dfscounter++ // v "discovered", number it
- for each edge (v,x)
  - if (x.dfs# = -1) // (x previously undiscovered)
    - DFS(x)
  - else …
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?* Using depth first search

---

DFS($v$) – Recursive version

**Global Initialization:**
for all nodes $v$, $v.dfs# = -1$ // mark $v$ "undiscovered"
dfscounter = 0

DFS($v$)
$v.dfs# = dfscounter++$ // $v$ "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

---

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.
Simple Case: Artic. Pts in a tree

Which nodes in a rooted tree are articulation points?

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 ....
Leaf is never an articulation point
non-leaf, non-root node $u$ is an articulation point

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: \( \text{LOW}(v) \) 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.
Articulation Points:  
the "LOW" function

Definition:  \( \text{LOW}(v) \) 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.  

\( v \) (non-root) articulation point iff some child \( x \) of \( v \) has \( \text{LOW}(x) \geq \text{dfs#}(v) \)

Articulation Points:  
the "LOW" function

Definition:  \( \text{LOW}(v) \) 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.  

\( v \) (non-root) articulation point iff some child \( x \) of \( v \) has \( \text{LOW}(x) \geq \text{dfs#}(v) \)

\[ \text{LOW}(v) = \min \left( \{ \text{dfs#}(v) \} \cup \{ \text{LOW}(w) | w \text{ a child of } v \} \cup \{ \text{dfs#}(x) | \{v,x\} \text{ is a back edge from } v \} \right) \]

DFS(v) for Finding Articulation Points

Global initialization: \( v.\text{dfs#} = -1 \) for all \( v \).  

DFS(v)  
\[ v.\text{dfs#} = \text{dfscounter}++ \]  
\[ v.\text{low} = v.\text{dfs#} \] // initialization  

for each edge \( \{v,x\} \)  
if (\(x.\text{dfs#} == -1\)) // \( x \) is undiscovered  
\[ \text{DFS}(x) \]  
\[ v.\text{low} = \min(v.\text{low}, x.\text{low}) \]  
if (\(x.\text{low} >= v.\text{dfs#}\))  
\quad print \"v is art. pt., separating x\"  
\quad \text{Equiv: \"if \( \{v,x\} \) is a back edge\"}  
else if (\(x \) is not \( v \)'s parent)  
\quad v.\text{low} = \min(v.\text{low}, x.\text{dfs#})  
\quad \text{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