Graphs
Objects & Relationships

Facebook friends:
  Obj: People
  Rel: Two are related if they are friends

Cities and Roads:
  Obj: Cities
  Rel: Two are related if they have a road between them

Data flow in programs:
  Obj: Lines of the program
  Rel: Two are related if one line depends on the other
Graphs

Objects: "vertices," aka "nodes"

Relationships between pairs: "edges”

Formally, a graph \( G = (V, E) \) is a pair of sets, \( V \) the vertices and \( E \) the edges. Each edge is a set or tuple of two vertices.
Undirected Graph $G = (V, E)$
Undirected Graph \( G = (V,E) \)
Undirected Graph $G = (V, E)$
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 $G = (V, E)$
Directed Graph $G = (V, E)$
Directed Graph $G = (V, E)$
Directed Graph $G = (V,E)$
Graphs

**Degree of a vertex, **$\text{deg}(v)$**: \# edges that touch that vertex

$\text{deg}(6) = 3$.

**Path**: sequence of distinct vertices s.t. each vertex is connected to the next vertex with an edge

Eg: 3, 6, 5, 4
**Connected**: Graph is connected if there is a path between every two vertices

**Connected component**: Maximal set of connected vertices

**Cycle**: Path of length > 1 that has the same start and end. Eg: 6,5,7

**Tree**: A connected graph with no cycles
# Vertices vs # Edges

Let G be an undirected graph with $n$ vertices and $m$ edges. How are $n$ and $m$ related?
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:

$$0 \leq m \leq \frac{n(n-1)}{2} = \mathcal{O}(n^2)$$
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: \( n+m = O(n^2) \), but \( n+m \) usually way better!
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:

\[
\begin{array}{cccc}
A & 7 & 3 & 4 \\
\hline
A & 0 & 0 & 1 & 1 \\
7 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 \\
4 & 1 & 1 & 1 & 0 \\
\end{array}
\]
Representing Graph $G = (V, E)$

Vertex set $V = \{v_1, \ldots, v_n\}$

Adjacency Matrix $A$

$A[i,j] = 1$ iff $(v_i, v_j) \in E$

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

$m \ll n^2$
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
BFS(v)
BFS(v)

Queue: 2 3
BFS(v)

Queue: 3 4
BFS(v)

Queue: 4 5 6 7

Diagram of a BFS traversal with nodes labeled 1 to 13.
BFS(v)

Queue: 5 6 7 8 9
BFS(v)

Queue: 8 9 10 11
BFS(v)

Queue: 10 11 12 13
BFS(v)

Queue:

[Diagram of a graph with nodes labeled 1 to 13, showing the breadth-first search traversal order and the queue.]
BFS: Analysis, 1

Global initialization: mark all vertices "undiscovered"

\[ \text{BFS}(s) \]

- \(O(n)\) mark \( s \) "discovered"
- \(O(1)\) queue = \{ \( s \) \}
- \(O(n)\) while queue not empty
  - \(O(n)\) \( u = \text{remove}\_first(\text{queue}) \)
    - for each edge \{\( u, x \)\}
      - if (\( x \) is undiscovered)
        - \( \text{mark } x \text{ discovered} \)
        - \( \text{append } x \text{ on queue} \)
  - \( \text{mark } u \text{ fully explored} \)

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

\[ = O(n^2) \]
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 $O(n+m)$, $n = \# \text{ nodes}$, $m = \# \text{ edges}$
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.

not true of every spanning tree!
Proof of correctness

Lemma 1: Every vertex at level $i$ is explored after every vertex at level $i-1$.

Proof is by induction on $i$.
Base case: $i = 1$. True.
Induction step: Let $u$ be at level $i$, and $v$ be at level $i-1$. Since we use a queue, it is enough to prove that $u$ is added to the queue after $v$. But $u$ was added when a vertex at level $i-1$ was explored, and $v$ is added when a vertex of level $i-2$ was explored. So $u$ is added after $v$ by induction.
Proof of correctness

Lemma 2: Level \( i \) in this tree are exactly those vertices \( u \) such that the shortest path (in \( G \), not just the tree) from the root is of length \( i \).

Proof is by induction on \( i \).
Base case: \( i = 0 \). True.

Induction step: Every vertex \( u \) at level \( i \) certainly has distance at most \( i \), because we discover a path of length \( i \) from \( u \) to \( v \). If the distance from the root is less than \( i \), and \( u \) was discovered when exploring \( v \) (at level \( i-1 \)), then \( u \) is a neighbor of a vertex \( b \) at distance (and level) \( < i-1 \). But then, by Lemma 1, \( b \) would have been explored before \( v \), and \( u \) would have been added in level \( i-1 \).
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.

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

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.

**Lemma:**
al 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

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

"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.

![a bipartite graph $G$](image1)

![another drawing of $G$](image2)
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* (2-colorable)  
*not bipartite* (not 2-colorable)  
*not bipartite* (not 2-colorable)
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).
Bipartite Graphs

Lemma. Let $G$ be a connected graph, and let $L_0$, ..., $L_k$ 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.

Case (i)
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.

![Bipartite bipartite](image1)

*bipartite* (2-colorable)

![Not bipartite not bipartite](image2)

*not bipartite* (not 2-colorable)
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 is "=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., \(\forall\) 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 \( (v_j, v_i) \) is an edge and \( v_1, \ldots, v_n \) is a topological order, we must have \( j < i \), a contradiction.

If all edges go L→R, you can't loop back to close a cycle.
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?
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.
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:
Topological Ordering Algorithm: Example

Topological order: $v_1$
Topological Ordering Algorithm: Example

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

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: $v_1, v_2, v_3, v_4, v_5$
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

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 go. 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"
Suppose edge lists at each vertex are sorted alphabetically.
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
(Edge list)
A (B,J)
B (A,C,J)
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

- **A,1**
- **B,2**
- **C,3**
- **D**
- **E**
- **F**
- **G**
- **H**
- **I**
- **J**
- **K**
- **L**
- **M**

Color code:
- undiscovered
- discovered
- fully-explored

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

- **G**
- **H**
- **I**
- **K**
- **L**
- **M**
Suppose edge lists at each vertex are sorted alphabetically.

DFS(A)

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

Color code:
- undiscovered
- discovered
- fully-explored
**DFS(A)**

Suppose edge lists at each vertex are sorted alphabetically.

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

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

Diagram:
- A (1)
- B (2)
- C (3)
- D (4)
- E (5)
- G
- H
- I
- J
- K
- L
- M
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Call Stack:
(Edge list)

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Edge List</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(B, J)</td>
</tr>
<tr>
<td>B</td>
<td>(A, C, J)</td>
</tr>
<tr>
<td>C</td>
<td>(B, D, G, H)</td>
</tr>
<tr>
<td>D</td>
<td>(C, E, F)</td>
</tr>
<tr>
<td>E</td>
<td>(D, F)</td>
</tr>
<tr>
<td>F</td>
<td>(D, E, G)</td>
</tr>
</tbody>
</table>
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Color code:
- undiscovered
- discovered
- fully-explored

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

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

**DFS(A)**

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)
  - D (G,E,F)
  - E (D,F)
  - F (D,E,G)
  - G (C,F)
Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

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

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

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

A,1
B,2
C,3
D,4
E,5
F,6
G,7
H
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 (B, J)
- B (A, C, J)
- C (B, D, G, H)
- D (E, F, I)

**DFS(A)**
DFS(A)

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)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)

Diagram:
- A,1
- B,2
- C,3
- D,4
- E,5
- G,7
- F,6
- H
- K
- L
- M

Vertices are shaded according to their status:
- Green for undiscovered
- Red for discovered
- Black for fully-explored
Suppose edge lists at each vertex are sorted alphabetically.

DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)
  - H (C,I,J)
  - D
  - E
  - F
  - G
  - I
  - J
  - K
  - L
  - M
Suppose edge lists at each vertex are sorted alphabetically.
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

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

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

**DFS(A)**

Call Stack:
- A (B,J)
- B (A,C,J)
- C (B,D,G,H)
- H (C,I,J)

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

**DFS(A)**

Call Stack:
- (Edge list)
  - A (J, B)
  - B (A, J, C)
  - C (B, D, G, H)
  - H (C, I, J)
  - J (A, B, H, K, L)

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

**DFS(A)**

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
  - A (B,J)
  - B (A,C,J)
  - C (B,D,G,H)
  - H (C,I,J)
  - J (A,B,H,K,L)
  - K (J,L)
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

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

Call Stack:
- (Edge list)
- A (B,J)
- B (A,C,J)
- C (B,D,G,H)
- H (C,I,J)
- J (A,B,H,K,L)
- K (J,L)
- L (J,K,M)
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.

Call Stack:
(Edge list)
A (B, J)
B (A, C, J)
C (B, D, G, H)
H (C, I, J)
J (A, B, H, K, L)
K (J, L)
L (J, K, M)

Color code:
- undiscovered
- discovered
- fully-explored
**DFS(A)**

Suppose edge lists at each vertex are sorted alphabetically.

**Call Stack:**

- A (B, J)
- B (A, C, J)
- C (B, D, G, H)
- H (C, I, J)
- J (A, B, H, K, L)
- K (J, L)

**Color code:**
- undiscovered
- discovered
- fully-explored
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.

Call Stack:
(Edge list)
A (B,J)
B (A,C,J)
C (B,D,G,H)
H (C,I,J)
J (A,B,H,K,L)

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

**DFS(A)**

Call Stack:
(Edge list)

A (B, J)
B (A, C, J)
C (B, D, G, H)
H (C, I, J)
J (A, B, H, K, L)
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

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

A, 1
B, 2
C, 3
D, 4
E, 5
F, 6
G, 7
H, 8
I, 9
J, 10
K, 11
L, 12
M, 13
Suppose edge lists at each vertex are sorted alphabetically.

DFS(A)

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

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

E, 5
F, 6
G, 7
H, 8
I, 9
J, 10
K, 11
L, 12
M, 13
D, 4
C, 3
B, 2
A, 1

DFS(A)

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)
- A (B, J)
- B (A, C, J)

Diagram:
- Color codes for vertices:
  - A: undiscovered
  - B: discovered
  - C: fully-explored
DFS(A)

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
- (Edge list)
- A (B, J)
Suppose edge lists at each vertex are sorted alphabetically.

**DFS(A)**

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
*(Edge list)*
- A (B, J)
**DFS(A)**

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
(Edge list)

TA-DA!!
DFS(A)

Edge code:
- Tree edge
- Back edge
DFS(A)
DFS(A)

Edge code:
- Tree edge
- Back edge

A,1
B,2
C,3
D,4
E,5
G,7
F,6
H,8
I,9
J,10
K,11
L,12
M,13
DFS(A)

Edge code:
- Tree edge
- Back edge

DFS(A)

Edge code:
- Tree edge
- Back edge

Diagram showing a directed graph with nodes labeled A to M, each with a number from 1 to 13. The edges are indicated by solid and dashed lines, representing tree edges and back edges respectively.
DFS(A)

Edge code:
- Tree edge
- Back edge
DFS(A)

Edge code:
- Tree edge
- Back edge
- No Cross Edges!
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
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?**
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