CSE 421

Depth First Search

Yin Tat Lee
Summary of last lecture

- **BFS**($s$) implemented using queue.

- Edges into then-undiscovered vertices define a tree – the “Breadth First spanning tree” of $G$

- Level $i$ in the tree are exactly all vertices $v$ s.t., the shortest path (in $G$) from the root $s$ to $v$ is of length $i$

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

- Applications:
  - Shortest Path
  - Connected component
  - Test bipartiteness / 2-coloring
Preview of this lecture

• Depth First Search
• 1 property: non-tree edge is vertical instead of horizontal
• 1 application: topological sort
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(s) – Recursive version

**Initialization**: mark all vertices undiscovered

DFS(\(v\))
- Mark \(v\) discovered

for each edge \(\{v, x\}\)
  - if (\(x\) is undiscovered)
    - Mark \(x\) discovered
    - DFS(\(x\))

Mark \(v\) fully-discovered
Non-Tree Edges in DFS

BFS tree ≠ DFS tree, but, as with BFS, DFS has found a tree in the graph s.t. non-tree edges are "simple" in some way.

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

Suppose edge lists at each vertex are sorted alphabetically.

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack (Edge list):
A (B,J)

st[] = {1}
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

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

st[] = \{1,2\}
DFS(A)

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

st[] = 
{1,2,3}

Color code:
undiscovered
discovered
fully-explored
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)

st[] = {1,2,3,4}
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)

st[] = {1, 2, 3, 4, 5}
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

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

st[] = {1, 2, 3, 4, 5, 6}
DFS(A)

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)

st[] =
{1, 2, 3, 4, 5, 6, 7}
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 (E,F)
- E (D,F)
- F (D,E,G)
- G (C,F)

st[] = {1, 2, 3, 4, 5, 6, 7}
DFS(A)

Call Stack:
(Edge list)

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

st[] =
{1,2,3,4,5,6}
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)

\( st[] = \{1,2,3,4,5\} \)
DFS(A)

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

\[\text{st[]} = \{1, 2, 3, 4\}\]
DFS(A)

Color code:
undiscovered
discovered
fully-explored

Call Stack:
(Edge list)

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

st[] =
{1,2,3}
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)

st[] = {1,2,3,8}
DFS(A)

Call Stack:
(Edge list)

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

st[] = {1,2,3,8,9}
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)

st[] = {1,2,3,8}
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)

st[] = {1, 2, 3, 8, 10}
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)

st[] = {1, 2, 3, 8, 10, 11}
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)

st[] = {1, 2, 3, 8, 10, 11, 12}
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)
- M (L)

\[ st[] = \{1,2,3,8,10,11,12,13\} \]
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)

st[] = {1,2,3,8,10,11,12}
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)

st[] =
{1,2,3,8,10,11}
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)

\[ st[] = \{1, 2, 3, 8, 10\} \]
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)

st[] =
{1,2,3,8,10}
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)

st[] = {1,2,3,8}
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

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

\[ st[] = \{1, 2, 3\} \]
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

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

st[] = {1,2}
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

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

st[] = {1,2}
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
(Edge list)

A (B, J)

st[] = {1}
**DFS(A)**

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

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

<table>
<thead>
<tr>
<th>Node</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>discovered</td>
</tr>
<tr>
<td>B</td>
<td>undiscovered</td>
</tr>
<tr>
<td>C</td>
<td>undiscovered</td>
</tr>
<tr>
<td>D</td>
<td>undiscovered</td>
</tr>
<tr>
<td>E</td>
<td>undiscovered</td>
</tr>
<tr>
<td>F</td>
<td>undiscovered</td>
</tr>
<tr>
<td>G</td>
<td>undiscovered</td>
</tr>
<tr>
<td>H</td>
<td>undiscovered</td>
</tr>
<tr>
<td>I</td>
<td>undiscovered</td>
</tr>
<tr>
<td>J</td>
<td>undiscovered</td>
</tr>
<tr>
<td>K</td>
<td>undiscovered</td>
</tr>
<tr>
<td>L</td>
<td>undiscovered</td>
</tr>
<tr>
<td>M</td>
<td>undiscovered</td>
</tr>
</tbody>
</table>

```
st[] = {1}
```
DFS(A)

Color code:
- undiscovered
- discovered
- fully-explored

Call Stack:
(Edge list)
TA-DA!!

st[] = {}
DFS(A)

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

Edge code:
- Tree edge
- Back edge
- No Cross Edges!
Properties of (undirected) DFS

Like BFS($s$):
• DFS($s$) visits $x$ iff there is a path in $G$ from $s$ to $x$
  So, we can use DFS to find connected components
• 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
Non-Tree Edges in DFS

**Lemma:** For every edge \( \{x, y\} \), if \( \{x, y\} \) is not in DFS tree, then one of \( x \) or \( y \) is an ancestor of the other in the tree.

**Proof:**

Suppose that \( x \) is visited first.

Therefore \( \text{DFS}(x) \) was called before \( \text{DFS}(y) \)

Since \( \{x, y\} \) is not in DFS tree, \( y \) was visited when the edge \( \{x, y\} \) was examined during \( \text{DFS}(x) \)

Therefore \( y \) was visited during the call to \( \text{DFS}(x) \) so \( y \) is a descendant of \( x \).
DAGs and Topological Ordering
Precedence Constraints

In a directed graph, an edge \((i, j)\) means task \(i\) must occur before task \(j\).

Applications

- Course prerequisite:
  
  course \(i\) must be taken before \(j\)

- Compilation:
  
  must compile module \(i\) before \(j\)

- Computing overflow:
  
  output of job \(i\) is part of input to job \(j\)

- Manufacturing or assembly:
  
  sand it before paint it
Directed Acyclic Graphs (DAG)

Def: A DAG is a directed acyclic graph, i.e., one that contains no directed cycles.

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 $i < j$.
Lemma: If $G$ has a topological order, then $G$ is a DAG.

Proof. (by contradiction)
Suppose that $G$ has a topological order $1, 2, \ldots, n$ and that $G$ also has a directed cycle $C$.
Let $i$ be the lowest-indexed node in $C$, and let $j$ be the node just before $i$; thus $(j, i)$ is an (directed) edge.
By our choice of $i$, we have $i < j$.
On the other hand, since $(j, i)$ is an edge and $1, 2, \ldots, n$ is a topological order, we must have $j < i$, a contradiction.

\[ \text{the supposed topological order: } 1, 2, \ldots, n \]
DAGs: A Sufficient Condition

G has a topological order

? 

G is a DAG
Every DAG has a source node

**Lemma:** If \( G \) is a DAG, then \( G \) has a node with no incoming edges (i.e., a source).

**Proof.** (by contradiction)
Suppose that \( G \) is a DAG and it has no source
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.

The proof is similar to “tree has \( n - 1 \) edges”.
**DAG => Topological Order**

**Lemma:** If $G$ is a DAG, then $G$ has a topological order

**Proof.** (by induction on $n$)

**Base case:** true if $n = 1$.

**Hypothesis:** Every DAG with $n - 1$ vertices has a topological ordering.

**Inductive Step:** Given DAG with $n > 1$ nodes, find a source node $v$.

$G - \{v\}$ is a DAG, since deleting $v$ cannot create cycles.

By 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.

**Reminder:** Always remove vertices/edges to use IH
A Characterization of DAGs

G has a topological order $\iff$ G is a DAG
Topological Order Algorithm 1: Example
Topological Order Algorithm 1: Example

Topological order: 1, 2, 3, 4, 5, 6, 7